Lambda-Upsilon-Omega the 1989 cookbook

Philippe Flajolet, Paul Zimmermann, Bruno Salvy

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

Rapports de Recherche

N°1073

Programme 2

Calcul symbolique, Programmation
et Génie logiciel

LAMBDA-UPSILON-OMEGA
THE 1989 COOKBOOK

Philippe FLAJOLET
Bruno SALVY
Paul ZIMMERMANN

Août 1989

Lambda–Upsilon–Omega, The 1989 CookBook

Philippe Flajolet
INRIA Rocquencourt,
F-78150 Le Chesnay (France)

Bruno Salvy
INRIA and LIX, Ecole Polytechnique,
F-91128 Palaiseau (France)

Paul Zimmermann
INRIA Rocquencourt

Abstract. *Lambda–Upsilon–Omega ($\Lambda\Upsilon\Omega$) is a research tool designed to assist the average case analysis of some well defined classes of algorithms and data structures. This cookbook consists of an informal introduction to the system together with eighteen examples of programmes that are automatically analyzed.*

Amongst the applications treated here, we find: addition chains, quantitative concurrency analysis of simple systems, symbolic manipulation algorithms such as formal differentiation, simplification and rewriting systems, as well as combinatorial models including various tree and permutation statistics and functional graphs with applications to integer factorisation.

Lambda–Upsilon–Omega, Livre de Recettes 1989

Résumé. Lambda–Upsilon–Omega ($\Lambda\Upsilon\Omega$) est un outil de recherche conçu pour aider à l'analyse automatique de classes bien définies d'algorithmes et de structures de données. Ce "livre de recettes" contient une présentation informelle du système accompagnée de dix–huit exemples de programmes analysés automatiquement.

Parmi les applications traitées, l'on trouve : des heuristiques pour les chaînes d'addition, une analyse quantitative de systèmes concurrents de forme simple, des algorithmes de manipulation symbolique tels la dérivation formelle, la simplification et certaines règles de réécriture, ainsi que des modèles combinatoires qui incluent diverses statistiques sur les arbres, les permutations et les graphes fonctionnels avec application à la factorisation d'entiers.

The $\Lambda\Upsilon\Omega$ system

I Introduction

Lambda–Upsilon–Omega[1], $\Lambda\Upsilon\Omega$ for short, is a system designed to perform automatic average–case analysis of well defined classes of algorithms and data structures. This report consists in a brief description of the scope of the system followed by a collection of some twenty or so examples of application.

$\Lambda\Upsilon\Omega$ attacks the analysis of an important class of algorithms that operate over *decomposable* data structures. In short, a data type (or "type") can be specified by means of a few well recognised set–theoretic constructions, including union of types, formation of Cartesian products ("records" in classical programming languages), formation of sequences (lists), sets, and multisets. A type specification can be either explicit or recursive.

An algorithm is then specified in the system by means of an *Algorithm Description Language* (Adl). Adl is a language whose primitives correspond closely to the type structuring mechanisms.

For instance, one can define a recursive data type '`expression`' recursively as follows:

```
type expression = zero | one | x
                | product(plus,expression,expression)
                | product(times,expression,expression)
                | product(expo,expression);
 plus,times,expo,zero,one,x = atom(1);
```

In other words, a mathematical formula like

$$f = (e^{x \cdot e^x + x \cdot x} + 1),$$

which corresponds to the Lisp–like form

```
(+ (expo (+ (* x (expo x)) (* x x))) 1)
```

is thus of type `expression`.

We can specify in Adl any algorithm which is allowed to traverse components of Cartesian products, sets or sequences, perform tests by cases for types defined as unions of types, and possibly call itself or some other similar procedures recursively. For instance a raw *differentiation* procedure, as typically encountered in computer algebra systems, is rendered by the text that follows. (In $\Lambda\Upsilon\Omega$, we allow ourselves to write `f(u,v)` as a synonym for the Lisp form `(f u v)`.)

```
function diff(e : expression) : expression;
case e of
  plus(e1,e2)     : plus(diff(e1),diff(e2));
  times(e1,e2)    : plus(times(diff(e1),e2),
                         times(e1,diff(e2)));
  expo(e1)        : times(diff(e1),e);
  zero            : zero;
  one             : zero;
```

[1] The name derives from the Greek word $\lambda\acute{\upsilon}\omega$ meaning —amongst other things— 'I solve'. That verb is the root of the word 'analysis'.

```
    x                : one
end;
```

In $\Lambda\Upsilon^\Omega$, we can specify an arbitrary *time complexity measure* for each function called. Assume for instance that we count a cost of 1 each time we need to generate one of the atomic symbols `plus`, `times`, `expo`, `zero`, `one`. By default, the argument passing mechanism of $\Lambda\Upsilon^\Omega$ is by reference; then the worst case complexity of `diff` is $O(n)$, where n is the size of the `expression` input to `diff`.

Though its order is *a priori* known, the average case complexity is harder to find, when precise constants need to be determined: the programme's behaviour depends essentially on the relative frequencies of the atomic symbols in random expressions. First, $\Lambda\Upsilon^\Omega$ needs to determine—at least asymptotically—the number of possible expressions of size n. This number is found to be of the form

$$\text{expression}_n \sim C\frac{A^n}{\sqrt{\pi n^3}}$$

for some algebraic constants A and C that are explicitly determined. This provides the underlying statistical model of use.

Next, $\Lambda\Upsilon^\Omega$ proceeds to determine the average case complexity, again in asymptotic form. In this case it produces a result which, even after simplifications, is by no means trivial:

```
                                      1/2        1/2
av_tau_diff_n :=  1/23 n (35 - 6    ) + O(n    )

Floating point evaluation :
                                                  1/2
                                 1.415239576  n + O(n    )
```

We have thus found: *When applied to a random expression of size n, the differentiation algorithm* `diff` *has average case complexity* $1.415239576n + O(\sqrt{n})$.

We should now provide a few indications on the automatic analysis process that takes place inside $\Lambda\Upsilon^\Omega$'s entrails.

The $\Lambda\Upsilon^\Omega$ system is based on the conjunction on two ideas. First, extending some recent methodologies in combinatorial analysis, we find general correspondences between data structure and algorithm specifications on the one hand, and equations over *generating functions* on the other hand. Such correspondences can be applied automatically provided programmes to be analyzed fall into a well–defined category of "decomposable programmes" over "decomposable data structures". Computing automatically these generating function equations is achieved by the *Algebraic Analyzer* —ALAS— of the system which is implemented in the CAML dialect of ML, and has about 2,000 instructions.

Second, these generating function equations are passed to a collection of computer algebra routines written in the Maple language [CGG+88]. For equations that can be effectively solved (a "Solver" programme takes care of this interface), the generating functions are processed by the *Analytic Analyzer* —ANANAS. ANANAS is a collection of computer algebra routines written in the Maple language and comprising at present about 5,000 instructions. The purpose of this subsystem is to extract the *asymptotic form* of coefficients of generating functions —in the current stage of the system given in an *explicit form*. The analytic (asymptotic) analyzer implements a collection of powerful strategies based on complex analysis (singularity analysis and saddle point methods).

In essence, any programme specified in the kernel of $\Lambda\Upsilon^\Omega$ can be analyzed automatically in terms of generating functions, since the rules are *complete* with respect to the language. The full analysis process can thus fail at a later stage on two counts: Either the Solver may fail to obtain explicit expressions (no explicit expression may even exist) or the Analytic Analyzer may fail. However, as demonstrated by the collection of automatic analyses given here, a non negligible fraction of problems of interest can already be processed by this chain.

Finally, it is important to note that many current limitations of the system do not seem to be intrinsic. For instance, certain methods are known to deal with implicitly defined functions and preliminary investigations indicate that the gaps between various asymptotic methods could be filled. We shall briefly return to these subjects in Section VII.

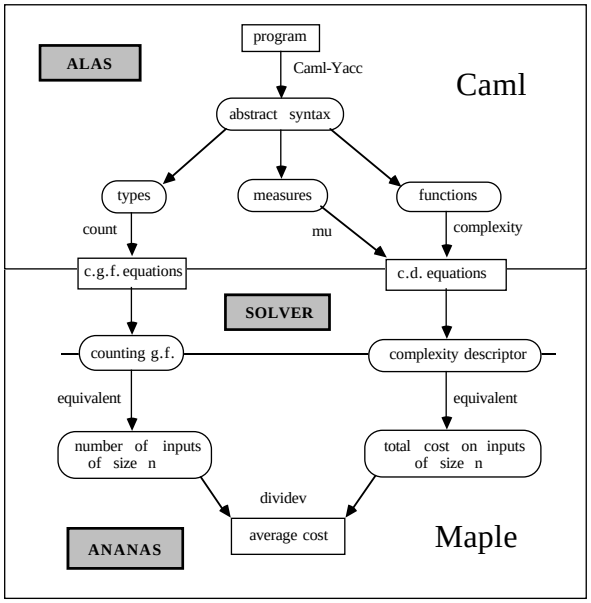


Figure 1: The internal structure of the $\Lambda\Upsilon^{\Omega}$ system

II A Sample Session

We shall illustrate the analysis process using $\Lambda\Upsilon^{\Omega}$ by a simple example, namely the computation of an 'exponential' x^c in a group structure G (e.g. the integers modulo a prime), using the standard "binary" method. This operation is of recognized importance in computational number theory and in cryptography (arithmetic public key systems).

II.1 Problem specification

Our starting point might be a programme in some language that implements the method which is based on the simple equations:

$$\begin{aligned} x^{2e} &= (x^e)^2 && (R0) \\ x^{2e+1} &= (x^e)^2 \cdot x\,. && (R1) \end{aligned}$$

For instance, a direct implementation in the Maple system for computer algebra is the following.

```
expmod:=proc(x,c)
# computes x^c in a group structure by squaring (.^2) and multiply (.*.)
# iquo(a,b)==a div b is the integer quotient function
        if c=0          then    1
        elif c mod 2 =0 then    c1:=iquo(c,2); (expmod(x,c1))^2
        else                    c1:=iquo(c,2); (expmod(x,c1))^2*x
        fi;
end;
```

The analysis problem consists in determining the expected number of multiplications and squaring operations that are performed in G when the exponent c is a random integer whose binary representation has length n.

To get some idea of what to expect, let us first do the (easy) analysis "by hand" . We perform n squaring operations, and one multiplication each time we encounter a 1 in the (binary representation of) exponent c. In total, we thus perform $\sim \frac{3}{2}n$ multiplications, since a random string of length n has on the average $\frac{1}{2}n$ bits

equal to 1. Obviously, the analysis depends only on the structure of exponent c and *not* on the particular value assigned to x.

II.2 Source Program (Adl)

The $\Lambda\Upsilon^{\Omega}$ specification of the algorithm is simple. It is done in the specific format called *Algorithm Description Language* or *Adl*.

```
type chain = sequence(bit);
     bit = zero | one;
     zero,one = atom(1);

procedure expmod(c:chain);
case c of
  () : nil;
  (zero,c1) : begin expmod(c1); squaring; end;
  (one,c1)  : begin expmod(c1); squaring; multiply; end;
end;

measure multiply : 1; squaring: 1;
to_analyze : expmod;
```

We declare the type `chain` as being an arbitrary sequence of `bits`, a `bit` being either a `zero` or a `one`. The `zero` and `one` bits are basic building blocks of our combinatorial structures (`chains`), and we declare them as *atomic*. Each of them contributes 1 to the size of the resulting structure, a fact rendered by the `atom(1)` declaration. To simplify the discussion, in this preliminary example, we did not impose that a chain should have a leading `one` (a non zero integer always has a leading 1 in its binary representation).

We next specify procedure `expmod`. (The syntax of the language allows a call by pattern matching according to well defined rules given in Section III.) Finally, we specify the cost measure and the function to be analyzed. Here, a squaring or a multiplication are assigned unit cost, and the function to be analyzed is `expmod`.

As we see, the syntax is in the style of Pascal, with a sort of test by pattern matching in the style of ML.

II.3 Algebraic analysis

We need two quantities. We let $chain_n$ denote the possible number of `chains` of size n and let $\tau expmod_n$ denote the total number of operations (according to the assigned cost) effected by `expmod` when applied to all inputs of size n. The expected cost we are looking for is just

$$\frac{\tau\mathrm{expmod}_n}{\mathrm{chain}_n}$$

The analysis is done automatically by means of generating functions. We need to introduce the *counting generating function* associated to the inputs, and the *complexity descriptor* (generating function) associated to the costs, namely

$$\mathrm{chain}(z) = \sum_{n=0}^{\infty} \mathrm{chain}_n z^n \qquad \text{and} \qquad \tau\mathrm{expmod}(z) = \sum_{n=0}^{\infty} \tau\mathrm{expmod}_n z^n.$$

The *Algebraic Analyzer*'s task (ALAS) consists in computing these generating functions. It uses a set of formal translation rules from programmes to generating functions. We start with translating type definitions.

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.
```

```
Counting generating functions:
  chain(z)=Q(bit(z))
  bit(z)=zero(z)+one(z)
  zero(z)=z
  one(z)=z
```

For instance, the relation `chain(z)=Q(bit(z))`, where $Q(y) \equiv (1-y)^{-1}$, translates the type specification `chain=sequence(bit)`. The translation mechanisms obey a collection of classical rules from combinatorial analysis. (See Section IV and the exposition given by Vitter and Flajolet [VF89]).

The next stage consists in translating complexity descriptors that are generating functions of costs working inductively on the Adl programme structure.

```
Analyzing expmod:
  nil --> 0*1
  expmod(c1) --> zero(z)*tau_expmod(z)/1
  squaring --> 1*zero(z)*Q(bit(z))
  expmod(c1) --> one(z)*tau_expmod(z)/1
  squaring --> 1*one(z)*Q(bit(z))
  multiply --> 1*one(z)*Q(bit(z))

Complexity descriptors:
  tau_expmod(z)=0*1+zero(z)*tau_expmod(z)/1+1*zero(z)*Q(bit(z))+0+one(z)*tau_expmod(z)/1+1*one(z)*Q
(bit(z))+1*one(z)*Q(bit(z))+0+0
```

At this stage, the algebraic analyzer has completed its task and has produced a set of functional equations, here of a very simple form.

II.4 Solving equations

The interface between ALAS and ANANAS is ensured by the *Solver* programme (SOLVER) whose task is to try and derive explicit expressions for intervening generating functions.

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...
```

$$\texttt{chain(z)} = \frac{1}{1 - 2\,z}$$

$$\texttt{tau\_expmod(z)} = 3\,\frac{z^2}{(1 - 2\,z)^2}$$

The `chain` data type is explicitly defined so that the corresponding system of equations for generating functions is triangular and can be solved explicitly in terms of elementary functions.

The `expmod` procedure is recursive, so that its complexity descriptor is defined recursively in terms of itself. In this simple case, the equation is even a linear (first degree) equation that is readily solved.

We can check for instance that the number of `chains` of size n is the coefficient of z^n inside the generating function

$$\mathrm{chain}(z) = \frac{1}{1-2z},$$

which provides[2] $chain_n = [z^n](1-2z)^{-1} = 2^n$ as expected.

[2] We let as usual $[z^n]f(z)$ denote the coefficient of z^n in the Taylor expansion of $f(z)$. Thus, if $f(z) = \sum_n f_n z^n$, then $[z^n]f(z) = f_n$.

II.5 Asymptotic Analysis

The *Analytic Analyzer* (ANANAS) then takes these generating functions and attempts to extract automatically asymptotic information on the coefficients.

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Function expmod:
Number of inputs of expmod of size n is:
                                                 infinity
                                      n      (1/n)
                                     2  + O(---------------)
                                                   n
                                              (1/2)

Total cost for expmod on random inputs of size n is:
                                          (- 1 + n)            1
                                       3 2           n + O(--------)
                                                              n
                                                       (1/2)


Average cost for expmod on random inputs of size n is:

                                    av_tau_expmod_n := 3/2 n + O(1)

Floating point evaluation :

                                        1.500000000 n + O(1)
```

We found that the number of `chains` of size n satisfies asymptotically

$$\mathrm{chain}_n = 2^n + O\left(\frac{2^n}{n^\infty}\right),$$

where the error term means $O(2^n/x^M)$ for any $M > 0$. The error term is actually identically 0; its presence here is due to the fact that the analytic (asymptotic) analyzer treated the function as a general meromorphic function.

In the same vein, the complexity descriptor of `expmod` has coefficients found to satisfy

$$\tau\mathrm{expmod}_n = \frac{3}{2} n\, 2^n + O(2^n).$$

This quantity represents by definition the total complexity of `expmod` applied to all inputs (`chains`) of size n. Through a division, we finally conclude the analysis and obtain the *average case complexity* of procedure `expmod` in the form

$$\overline{\tau\mathrm{expmod}}_n = \frac{3}{2} n + O(1) = 1.500000000 n + O(1),$$

as was to be expected.

III The Algorithm Description Language

$\Lambda\Upsilon\Omega$ deals with simple data types that are constructed from a collection of data structuring mechanisms including union of types, Cartesian products, sequence and set formation. The programming primitives that it accepts are consistent with the data structuring mechanisms. They allow us to perform tests by cases and call procedures on (either all or specific) components of a structure. The task can be carried out in both plain ("unlabelled") or "labelled" universes.

III.1 Type Structuring Mechanisms

Plain Unlabelled Universe. A type specification is a collection of (possibly recursive) equations describing the way data types are constructed from basic *atomic objects*. Constructors can be of the following sort: *union, product, sequence, set, multiset, oriented cycle, non oriented cycle.*

A type description starts with the keyword `type`. The syntax corresponding to the basic constructors is

```
A = B | C ;
A = product(B1,B2) ;
A = sequence(B) ;
A = set(B) ;
A = multiset(B) ;
A = cycle(B) ;
A = ucycle(B) ;
```

Here, `product` represents a Cartesian product. The `sequence` construction in '`A=sequence(B);`' means that A is formed with all sequences of elements of class B,

$$A = \left\{ (b_1, b_2, \ldots, b_r) \mid r \geq 0,\ b_j \in B \right\}.$$

The declaration `A = set(B)` means that A is the class of all *finite* subsets of B. The declaration `A = multiset(B)` defines A as the class of all *finite* subsets with repetitions of B, where each element of B can appear several times. For example, if $B = \{x, y\}$, we will have $A = \left\{ \{x^i, y^j\} \,\middle|\, 0 \leq i, j \right\}$.

The declaration `A = cycle(B)` (resp. `A = ucycle(B)`) means that A is the class of oriented (resp. non oriented) cycles of elements of B. For instance, there are 11 oriented cycles of length 3 over $\{x, y, z\}$:

$$\{xxx, xxy, xxz, xyy, xyz, xzy, xzz, yyy, yyz, yzz, zzz\}$$

but only 10 non oriented cycles:

$$\{xxx, xxy, xxz, xyy, xyz, xzz, yyy, yyz, yzz, zzz\}$$

(the two oriented cycles xyz and xzy reduce to one non oriented cycle).

In general, the size of a particular structure is the total sum of the sizes of the atoms it contains. Atomic objects (letters in strings, nodes in trees and graphs) are usually declared of type '`atom(1);`', meaning that each of them contributes 1 to the size of the encapsulating structure. The empty structure `epsilon` of type `atom(0)` is also provided.

For instance, here are possible specifications for binary strings (`binstring1` and `binstring2`), binary trees (`bintree`) and general trees (`gentree`):

```
type binstring1 = sequence(a|b); % Iterative specification %
     a,b = atom(1);

     binstring2 = product(letter,binstring2) | epsilon; % Recursive spec. %
     letter = a | b; a,b = atom(1);

     bintree = leaf | product(node,bintree,bintree);    % node degrees = 0 or 2 %
     leaf = atom(0); node = atom(1);

     gentree = product(node,sequence(gentree));     % all node degrees allowed %
     node = atom(1);
```

These definitions also imply that the size of a string is its usual length (number of characters a or b), the size of a general tree is the total number of its nodes, and the size of binary tree is the number of its internal binary nodes (leaves do not contribute to size in this specification).

Labelled Universe. The set theoretic constructions above have *labelled* counterparts. The concept is borrowed from graphical enumerations and classical combinatorial analysis: In this context, each atom (of size 1)

is labelled with an integer; the collection of all labels of a structure constitutes an initial segment of the integers.

The product operation then means the *partitional product* of combinatorial analysis. If α and β are two labelled structures, then the product `product`(α, β) is obtained by performing all labellings of the pair (α, β) consistent with the order structure of α and β.

The sequence, set and cycle constructions are then defined from this partitional product. For instance, the class of all permutations `permu` can be specified as

```
type permu = set(cycl);
     cycl = cycle(elem);
     elem = Latom(1);
```

The indication that we are dealing with a labelled universe is provided by the declaration `Latom(1)`. The labelled property is then implicitly propagated upwards to the `cycl` and `permu` types.

III.2 Programming Primitives

The allowed programming primitives are: *sequencing, test–on–unions, partial component descent and full component iteration.* A notable feature of the language is that it has no assignment.

A procedure is specified in the language by

```
procedure P( a : A );     % A is a declared type, 'a' is an identifier %
begin
    <body>
end;
```

Procedures are the basic objects to be analyzed by $\Lambda\Upsilon^{\Omega}$. As in Pascal, a colon `a : A` means that a belongs to type A.

Sequencing is expressed like in Pascal by a semi–colon,

```
begin P ; Q end    % do P then Q %
```

and it has the usual meaning: "Execute programme fragment `P`, then execute programme fragment `Q`".

Test on Union is done by a 'casetype' in the style of a Pascal 'case'.

```
% Precondition: A = B | C; a : A; %
% B and C are named types %
casetype a of
       B : F{a};       % F is a programme segment involving a : A %
       C : G{a};
end;
```

Test of emptiness. For list-constructors, this is done by a 'case' which, like 'casetype', may be followed by an 'otherwise'.

```
% Precondition: A = sequence (B); a : A; %
case a of
      ()        : F;
      otherwise : G{a};
end;
```

The empty list is denoted by `()` and the empty set by `{}`.

Component Descent. This is used to go down into components of Cartesian products.

```
% Precondition: A = product(B,C); a : A; %
case a of
      (b,c) : H{b,c};  % H is a programme segment involving b : B and c : C %
end;
```

There is a similar construction for separating the first component of a sequence.

```
% Precondition: A = sequence(B); a : A; %
case a of
      (b,r) : H{b,c};  % H is a programme segment involving b : B and r : A %
end;
```

Basically, the idea is that if $a = (b_1, b_2, \ldots, b_k)$, then (b, r) matches with a giving $b = b_1$, the initial element, and $r = (b_2, \ldots, b_k)$, the "rest".
Random descent. This corresponds to going down in a random component of a list-constructor.

```
% Precondition: A = sequence(B); a : A; %
forone b in a do
      K{b}       % K is a programme segment involving b : B %
```

Component Iteration. This construction corresponds to iterating over *all* components of a sequence.

```
% Precondition: A = sequence(B); a : A; %
forall b in a do
      K{b}       % K is a programme segment involving b : B %
```

The syntax for set or cycle constructions are similar. All these constructions apply equally to plain unlabelled and labelled universes.
Procedure call. As in Pascal, the name of the procedure is followed by the argument(s) in parentheses. It is worth noting that *linearity* is required. Hence `f(u)` and `g(u,v)` are valid procedure calls, but not `h(u,u)`. Function composition is not allowed.

III.3 Complexity measures

To each identifier is associated a cost, either numeric or symbolic: The declaration

```
measure add : 1;
        mul : m;
```

enables us to count separately executions of `add` (with cost 1) and `mul` (with cost m). For instance, the procedure

```
type gentree=product(node,sequence(gentree));
        node=atom(1);

procedure treesize(t : gentree);
begin
        case t of
                (node,r): begin
                          count;
                          forall u in r do
                              treesize(u);
                          end;
         end;
end;
```

traverses tree t and applies the 'count' procedure each time it encounters a node. If we further specify

```
measure count : 1;
```

then the cost of `treesize` is exactly the size of its argument.

III.4 Extensions

The Algorithm Description Language provides a way of restricting *cardinality* of *list-constructors* (*sequence, set, cycle*). For instance, the declarations

```
A = sequence (B, card >= 1);
C = set (D, card = 7);
E = cycle (F, card <= 3);
G = set (H, card = odd);
```

imply that A is the class of lists of elements of B, with at least one element, that C is the class of subsets of 7 elements of D, that E is the class of cycles of at most 3 elements of F, and G is the class of subsets containing an odd number of elements of H.

The class of all partitions of an integer into 4 parts is thus specified as

```
type partition = multiset (number, card=4);
     number = sequence (one, card>=1);
     one = atom(1);
```

IV The Algebraic Analyzer —ALAS

Given a procedure specification '`procedure P(a:A);`' with companion type declarations, the task of the $\Lambda\Upsilon^{\Omega}$ system is to determine its average case complexity, when all objects in A are taken equally likely.

As we indicated earlier, in Section 2, one needs first to determine the *input statistics*, namely the number A_n of objects in A with a given size n:

$$A_n = \mathrm{card}\{\alpha \in A \mid |\alpha| = n\}. \tag{1a}$$

Next we need to determine the *cumulated complexity* of P over A_n. If $\tau P[\alpha]$ denotes the cost (according to the specified cost measure) of P applied to a particular $\alpha \in A$, the cumulated complexity is defined by

$$\tau P_n = \sum_{\substack{\alpha \in A \\ |\alpha| = n|}} \tau P[\alpha]. \tag{2a}$$

The final result to be produced by the system is the average cost,

$$\overline{\tau P_n} = \frac{\tau P_n}{A_n},$$

or at least its asymptotic form.

The analysis proceeds to determine this quantity using *generating functions*. First the *counting generating function* of input type A,

$$A(z) = \sum_{n \geq 0} A_n z^n, \tag{1b}$$

next the *complexity descriptor* of procedure P,

$$\tau P(z) = \sum_{n \geq 0} \tau P_n z^n. \tag{2b}$$

The task of the Algebraic Analyzer (see the example of Section 2) is to translate data type declarations such as

```
type chain = sequence(bit);
     bit = zero | one;
     zero,one = atom(1);
```

into equations for counting generating functions such as

$$\begin{aligned} \mathrm{chain}(z) &= 1/(1-\mathrm{bit}(z)) \\ \mathrm{bit}(z) &= \mathrm{zero}(z)+\mathrm{one}(z) \\ \mathrm{zero}(z) &= z \\ \mathrm{one}(z) &= z \end{aligned}$$

and to translate procedure and measure declarations

```
procedure expmod(c:chain);
case c of
  () : nil;
  (zero,c1) : begin expmod(c1); squaring; end;
  (one,c1)  : begin expmod(c1); squaring; multiply; end;
end;

measure multiply : 1; squaring: 1;
```

into equations for complexity descriptors

$$\begin{aligned} \tau\mathrm{expmod}(z) &= 0 \\ &+ \mathrm{zero}(z)\,\tau\mathrm{expmod}(z)+\mathrm{zero}(z)\,\mathrm{chain}(z) \\ &+ \mathrm{one}(z)\,\tau\mathrm{expmod}(z)+\mathrm{one}(z)\,\mathrm{chain}(z)+\mathrm{one}(z)\,\mathrm{chain}(z). \end{aligned}$$

Note that, at least in principle, these generating functions condense all the information necessary for obtaining average case complexity.

The translation task is achieved in 4 phases:

1. Conversion of the Adl programme into a syntax tree, and decomposition of this syntax tree into data type declarations and procedure declarations.
2. Translation of data type declarations into Caml objects representing counting generating functions.
3. Translation of procedure declarations into Caml objects representing complexity descriptors.
4. Conversion of all Caml objects for subsequent treatment by Maple.

In the Caml programming language, steps 1 and 4 are made easy by a built–in Yacc interface and a good pattern matching facility. We only discuss here Steps 2 and 3.

IV.1 Translation of data type declarations

Every symbol represents a data type, hence it translates into the corresponding generating function:

$$A \longrightarrow A(z)$$

Atomic declarations are easily translated, taking care of which universe we are in

$$\mathrm{atom}(k) \longrightarrow \begin{cases} z^k & \text{in an unlabelled universe} \\ z^k/k! & \text{in a labelled universe.} \end{cases}$$

A type specification is translated into a set of functional equations over generating functions, with type constructors being rendered by operators over formal power series.

In an unlabelled universe, some of the rules are

Disjoint Union	$\mathcal{A}=\mathcal{B}+\mathcal{C}$	$\Longrightarrow$	$a(z)=b(z)+c(z)$
Cartesian Product	$\mathcal{A}=\mathcal{B}\times\mathcal{C}$	$\Longrightarrow$	$a(z)=b(z)\;c(z)$
Sequence	$\mathcal{A}=\mathcal{B}^*$	$\Longrightarrow$	$a(z)=1/(1-b(z))$
PowerSet	$\mathcal{A}=2^{\mathcal{B}}$	$\Longrightarrow$	$a(z)=\exp(\Phi(b))$
MultiSet	$\mathcal{A}=\mathcal{M}(\mathcal{B})$	$\Longrightarrow$	$a(z)=\exp(\Psi(b))$

where the Pólya operators Φ and Ψ are defined by

$$\begin{aligned}\Phi(b) &= b(z)/1 - b(z^2)/2 + b(z^3)/3 - \cdots \\ \Psi(b) &= b(z)/1 + b(z^2)/2 + b(z^3)/3 + \cdots .\end{aligned}$$

In a labelled universe, ordinary generating functions of Eq. (1b,2b) are replaced by exponential generating functions[3], but the translation mechanisms are similar:

Disjoint Union	$\mathcal{A} = \mathcal{B} + \mathcal{C}$	$\Longrightarrow$	$\hat{a}(z) = \hat{b}(z) + \hat{c}(z)$
Labelled Product	$\mathcal{A} = \mathcal{B} * \mathcal{C}$	$\Longrightarrow$	$\hat{a}(z) = \hat{b}(z)\ \hat{c}(z)$
Labelled Sequence	$\mathcal{A} = \mathcal{B}^{<*>}$	$\Longrightarrow$	$\hat{a}(z) = 1/(1 - \hat{b}(z))$
Labelled Set	$\mathcal{A} = \mathcal{B}^{[*]}$	$\Longrightarrow$	$\hat{a}(z) = \exp(\hat{b}(z))$
Cycle	$\mathcal{A} = \mathcal{C}(\mathcal{B})$	$\Longrightarrow$	$\hat{a}(z) = \log 1/(1 - \hat{b}(z))$
Unordered Cycle	$\mathcal{A} = \mathcal{UC}(\mathcal{B})$	$\Longrightarrow$	$\hat{a}(z) = 1/2 \log(1/(1 - \hat{b}(z))) + \hat{b}(z)/2 + \hat{b}^2(z)/4.$

These rules derive from symbolic methods in combinatorial analysis (see Section VII).

IV.2 Translation of procedure declarations

A procedure P with arguments in a set A, (corresponding to a header '`procedure P(a:A);`') is translated into its complexity descriptor as defined in Eq. (2b):

$$P \longrightarrow \tau P(z).$$

From the specification of P (and its companion type declarations), we can build a set of translation rules that extend the symbolic approach of the previous section. For instance, a *program construction* corresponding to a descent into the component of a Cartesian product,

```
% Precondition: C = product(A,B);  procedure P(a : A) %
    procedure P(c : C);
    case c of
        (a,b) : Q(a);
    end;
```

translates into

$$\tau P(z) = \tau Q(z) B(z).$$

Observe that the translation of a programme segment involves both complexity descriptors and counting generating functions of intervening structures.

A collection of translation rules corresponding to programme constructs of Section 3 is given in the table below. We have adopted there the shorthand notation '$P : A$' in lieu of the ΛϒΩ statement '`procedure P(a:A);`'.

Unlabelled Universe	$\tau R(z)$
$P : \mathcal{A}, \quad Q : \mathcal{A}, \quad R \stackrel{\text{def}}{=} P; Q$	$\tau P(z) + \tau Q(z)$
$P : \mathcal{A}, \quad R : \mathcal{A} \cup \mathcal{B}, \quad R(c) \stackrel{\text{def}}{=}$ **if** $c \in \mathcal{A}$ **then** $P(c)$	$\tau P(z)$
$P : \mathcal{A}, \quad R : \mathcal{A} \times \mathcal{B}, \quad R((a,b)) \stackrel{\text{def}}{=} P(a)$	$\tau P(z) b(z)$
$P : \mathcal{A}, \quad R : \mathcal{A}^*, \quad R((a_1, \ldots, a_k)) \stackrel{\text{def}}{=} P(a_1); \ldots; P(a_k)$	$\tau P(z)/(1 - a(z))^2$
$P : \mathcal{A}, \quad R : 2^A, \quad R(\{a_1, \ldots, a_k\}) \stackrel{\text{def}}{=} P(a_1); \ldots; P(a_k)$	$\exp(\Phi(a))(\tau P(z) - \tau P(z^2) + \tau P(z^3) - \cdots)$
$P : \mathcal{A}, \quad R : MultiSet(\mathcal{A}), \quad R(\{a_1, \ldots, a_k\}) \stackrel{\text{def}}{=} P(a_1); \ldots; P(a_k)$	$\exp(\Psi(a))(\tau P(z) + \tau P(z^2) + \tau P(z^3) + \cdots)$

[3] If $\{f_n\}$ is a sequence of numbers, the associated ordinary generating function is $f(z) = \sum f_n z^n$, while the exponential generating function is $\hat{f}(z) = \sum_n f_n z^n/n!$.

Labelled Universe	$\hat{\tau}R(z)$
$P:\mathcal{A}, \quad R:\mathcal{A}*\mathcal{B}, \quad R((a,b)) \stackrel{\text{def}}{=} P(a)$	$\hat{\tau}P(z)\hat{b}(z)$
$P:\mathcal{A}, \quad R:\mathcal{A}^{<*>}, \quad R((a_1,\ldots,a_k)) \stackrel{\text{def}}{=} P(a_1);\ldots;P(a_k)$	$\hat{\tau}P(z)/(1-\hat{a}(z))^2$
$P:\mathcal{A}, \quad R:\mathcal{A}^{[*]}, \quad R(\{a_1,\ldots,a_k\}) \stackrel{\text{def}}{=} P(a_1);\ldots;P(a_k)$	$\hat{\tau}P(z)\exp(\hat{a}(z))$
$P:\mathcal{A}, \quad R:Cycle(\mathcal{A}), \quad R((a_1,\ldots,a_k)) \stackrel{\text{def}}{=} P(a_1);\ldots;P(a_k)$	$\hat{\tau}P(z)/(1-\hat{a}(z))$

Note. A user defined procedure is translated into a complexity descriptor by the rules above. A procedure that is left unspecified receives cost 0, if it has no explicit cost assignment in the `measure` part of the $\Lambda\Upsilon^{\Omega}$ programme, or a constant —either numeric or symbolic— cost otherwise. In this way, we can introduce an arbitrary number of counters in a program to be analyzed and specify any linear time complexity model.

IV.3 The Solver

The SOLVER is a piece of Maple code that lies at the interface between the algebraic analyzer ALAS and the analytic analyzer ANANAS. It processes the set of functional equations provided by ALAS, and passes them to Maple. In the process, elementary simplifications are carried out.

One of the tasks of the Solver is to choose the appropriate solution when several are possible. Returning to the specification of general trees in section III.2,

```
type gentree=product(node,sequence(gentree));
       node=atom(1);
```

we have a translation into counting generating functions:

$$\begin{aligned} gentree(z) &= node(z)Q(gentree(z)) \\ node(z) &= z. \end{aligned}$$

Thus $g(z) \equiv gentree(z)$ satisfies the equation

$$g(z) = \frac{z}{1-g(z)}.$$

That equation is a quadratic equation in disguise, and there are a *priori* two possible determinations:

$$\frac{1+\sqrt{1-4z}}{2} \quad \text{or} \quad \frac{1-\sqrt{1-4z}}{2}.$$

We should select the second form, because the first one has negative coefficients in its Taylor expansion. It is this form which will be passed to ANANAS for asymptotic processing of its coefficients.

The present state of the $\Lambda\Upsilon^{\Omega}$ system requires that generating functions should be solved explicitly in terms of elementary functions. This constraint should be somewhat relaxed in later versions of the system.

V The Analytic (Asymptotic) Analyzer —ANANAS

At this stage, our task is to take a generating function, defined either explicitly (for non recursive data types) or implicitly via a functional equation (for most recursive data types). The current version of $\Lambda\Upsilon^{\Omega}$ treats only functions that lead to explicit expressions after repeated usage of the 'solve' routine of Maple (see the SOLVER's description). We shall therefore limit ourselves to this case.

V.1 Analytic Principles

Let $f(z)$ be a function analytic at the origin. We assume further that $f(z)$ is explicitly given by an *expression*, a blend of sums, products, powers, exponential and logarithms. Most explicit generating functions constructed by the combinatorial tools of Section IV are of this type.

The starting point is Cauchy's coefficient formula

$$[z^n]f(z) = \frac{1}{2i\pi}\int_\Gamma f(z)\,\frac{dz}{z^{n+1}}, \tag{4.1}$$

where $[z^n]f(z)$ is the usual notation for the coefficient of z^n in the Taylor expansion of $f(z)$. The first fact one can deduce from this formula is that the behaviour of Taylor coefficients depends heavily on the singularities of $f(z)$. Several observations are useful here. First, functions defined by expressions are analytically continuable – except for possible isolated singularities – to the whole of the complex plane (though they may be multivalued). Second, by Pringsheim theorem, functions with positive coefficients (such is the case for our generating functions) always have a positive dominant singularity, a fact that eases considerably the search for singularities.

Once the singularities are known, two major classes of methods are applicable to determine the behaviour of the Taylor coefficients:

- For functions with singularities at a finite distance, the local behaviour of the function near its dominant singularities (the ones of smallest modulus) determines the growth order of the Taylor coefficients of the function. Asymptotic information is obtained by taking Γ to be a contour that comes close to the dominant singularities. Besides, when the singularity is of an 'algebraico-logarithmic' type (this means 'not too violent'), then by methods of the Darboux-Pólya type, one can translate a local expansion of the function around its dominant singularities to an asymptotic expansion of the coefficients.
- For entire functions and functions with essential singularities, saddle point contours Γ are usually applicable.

V.2 Algorithm

Though a complete algorithm covering all elementary functions is not (yet) available since the classification of singularities, even for such functions, is not fully complete, a good deal of functions arising in practice can be treated by the '`equivalent`' algorithm whose outline follows.

Procedure equivalent(f : expression) : expression;
{determines an asymptotic equivalent of $[z^n]f(z)$}

1. Determine whether $f(z)$ has singularities at a finite distance. If so, compute the smallest ones (which are dominant).
2. If $f(z)$ has finite singularities, compute a local expansion of $f(z)$ around its dominant singularity (-ies). This is called a singular expansion.
   - 2a. If a singular expansion is of an 'algebraico-logarithmic' type, then transfer singular expansions to coefficients.
   - 2b. If the function is large near its singularity, then apply saddle point methods like (3) below.
3. If $f(z)$ is entire, then use a saddle point integral.

It is important to note that a few *theorems*, whose conditions can be automatically tested, are used to support this algorithm.

Singularity Analysis. The classical form of the Darboux-Pólya method requires differentiability conditions on error terms. However, from [FO87], we now know that analytic continuation is enough to ensure the transition from (4.2) to (4.3), and by our earlier discussion, these conditions are always fulfilled for functions defined by expressions. Thus, the use of (2a) is guaranteed to be sound. Furthermore, that approach makes it possible to cope with singularities involving iterated logarithms as well (not yet implemented).

Saddle Point Integrals. There has been considerable interest for those methods, due to their recognised importance in mathematical physics and combinatorial enumerations. We thus know, from works by Hayman, Harris and Schoenfeld, or Odlyzko and Richmond, classes of functions *defined by closure properties* for which saddle point estimates are valid. Such conditions, that are extremely adequate for combinatorial generating functions, can be checked inductively on the expression.

V.3 Some Applications

Let us take here the occasion of a few examples to discuss some further features of ANANAS. The functions all arise as generating functions of (more or lesss natural) combinatorial structures.

(E1) Trees of cycles of necklaces of beads;

```
type bead = Latom(1);
     node = cycle(cycle(bead));
     tree = product(node,sequence(tree));
```

(E2) Involutive permutations;

```
type involution = set(cycle(elem,card<=2));
     elem = Latom(1);
```

(E3) Children's Rounds of [Sta78];

```
type rounds = set(product(child,cycle(child)));
     child = Latom(1);
```

(E4) "Military Rounds".

```
type rounds = sequence(product(soldier,cycle(soldier)));
     soldier = Latom(1);
```

Their treatment by the `equivalent` procedure of ANANAS illustrates the use of different asymptotic techniques in the system.

```
E1> equivalent(1/2*(1-sqrt(1-4*log(1/(1-log(1/(1-z)))))));
                     1/2
-1/2   exp(exp(-1/4))    exp(-3/8)
           3/2                                 1/2
      1 / (n    (- exp(exp(-1/4)) exp(-1) + 1)   )
                                           n   1/2
         / ((- exp(exp(-1/4)) exp(-1) + 1)  Pi   ) + etc ...
                                  (1/2 - n)
        (- exp(exp(-1/4) - 1) + 1)
   + O(-------------------------------------)
                         7/2
                        n
```

This example demonstrates the processing of functions with singularities at a finite distance. The singularity has an explicit form and the function behaves locally like a square root, whence the final result of the form:

$$Cn^{-3/2}\left(1-e^{e^{-1/4}-1}\right)^{-n}.$$

```
E2> equivalent(exp(z+z^2/2));
                            1/2                                1/2 2
exp(- 1/2 - 1/2 (1 + 4 n)    + 1/2 (- 1/2 - 1/2 (1 + 4 n)   ) )

                              1/2 n     1/2   1/2                         1/2 1/2
   / ((- 1/2 - 1/2 (1 + 4 n)   )  (-2)    Pi    (- 1/2 - 1/2 (1 + 4 n)   )   )

              1/4
   / (1 + 4 n)
```

This demonstrates the asymptotic analysis of the Involution numbers ([Knu73, p.65-67] does it by the Laplace method). It is treated here by "Hayman admissibility". Hayman's Theorem provides a class of admissible functions for which a saddle point argument (Step 3 of procedure '`equivalent`') can be applied: *If f and g are admissible, h is an entire function and P is a real polynomial with a positive leading coefficient, then*

$$\exp(f),\ f+P,\ P(f),\ \textit{and under suitable conditions, } f+h$$

are admissible. These conditions can be checked syntactically here.

```
E3> equivalent((1-z)^(-z),5);
           ln(n)     gamma      1     ln(n)          1     gamma       ln(n)
1 - 1/n - ------- - ------- + ---- - ------- + 5/2 ---- - ------- + O(--------)
             3         3       3        4           4        4           5
            n         n       n        n           n        n           n
```

This is a further illustration of singularity analysis in a non trivial case, here the difficulty lies in the singular expansion.

```
E4> equivalent(1/(1-z*ln(1/(1-z))));
                                    Let
s[   1   ]=   .7407536434
Root of    1+z*ln(1-z)   =0
                                then we get :

                           1                                  1
          ---------------------------------------- + O(----------)
                                                 2            n  2
              n                            s[1]          s[1]  n
           s[1]  (- s[1] ln(1 - s[1]) + ----------)
                                        1 - s[1]
```

In this last example, the dominant singularity cannot be found explicitly. However, the programme is able to perform the computations purely symbolically.

VI A Collection of Examples

Each of the chapters that follow contains the script [4] of an analysis obtained by the $\Lambda\Upsilon^{\Omega}$ system.

The examples can be organised into three groups:

Part A. Regular Languages and Finite Automata;

Part B. Context–Free Languages, Terms and Symbolic Manipulation Algorithms;

Part C. Combinatorial Problems.

In the first category, we find problems naturally described by finite automata and regular languages. The easy example of addition chains that we used in Section 2 is given *in extenso*. It is followed by an analysis of some heuristics for addition chains that have been used in primality testing. The next chapter in this part shows quantitative concurrency estimates for a system described by a finite automaton, and it is representative of a whole class of analyses that can be performed in $\Lambda\Upsilon^{\Omega}$. Part A concludes with a combinatorial problem related to the the counting of some special bipartite graphs. Common to all these cases is the occurrence of rational generating functions in the analyses.

The second category covers the simplest of all recursive data structures, namely trees. The underlying random tree models are useful for analyzing a variety of algorithms in *symbolic manipulation systems*.

[4] In a chapter, it is only the introductory section describing the problem that has been produced manually, the rest being generated automatically through a LaTeX interface.

We first examine a differentiation algorithm with results that were mentioned in the introduction. Next we consider the case where arguments are passed by "value" (copied) and, in that framework, we analyze higher order differentiation. The next two chapters deal with the analysis of some simplification algorithms (expansion through distributivity) or rewriting systems, and the last one is relative to some analyses suggested by studies in programme transformations.

The third category presents a selection of miscellaneous combinatorial and probabilistic problems. First, we examine some statistics on 2–regular graphs. Next, we consider a fairly complicated combinatorial structure, namely random *trains*. We continue with a heuristic analysis of Pollard's integer factorisation algorithm, which takes us into the realm of functional graphs and random mappings. Then, we analyze the variance of path length in trees, a problem on partitions, and a counting problem (dequeue sortable permutations) for which bounds can be derived using $\Lambda\Upsilon^{\Omega}$. We conclude with a complete treatment of Banach's celebrated matchbox problem.

VII Concluding Remarks

In this section, we sketch the position of the $\Lambda\Upsilon^{\Omega}$ system within the field of automatic analysis of algorithms, and discuss some further research trends in the area.

As far as the authors are aware, the oldest attempt to automatically analyze the complexity of (some) programmes goes back to Wegbreit in 1975 [Weg75]. Wegbreit was interested in a collection of simple list manipulation programmes and his system, METRIC, would compile programme specifications into recurrence equations describing average-case as well as other measures of programme complexity. Wegbreit's treatment uses Markovian approximations where complex tests are assumed to have a fixed probability of being satisfied,. In this way, equations are reduced to linear recurrences whose treatment is elementary, and there is no special need of a computer algebra system.

An approach somewhat similar in scope to Wegbreit's, but applied to worst-case analysis was developed by Le Métayer [Met88]. His system, ACE analyzes functional programmes using a fairly large (1,000) data base of rules that translate programme schemes into complexity equations. A related line of research is currently being explored by Wolf Zimmermann [Zim88b] who returned to average case analysis.

In the mean time, Kozen [Koz81] developed a semantics view of probabilistic programmes, where programmes are viewed as linear operators on Banach spaces. Ramshaw, in his thesis [Ram79], developed a logical framework for verifying complexity assertions on programmes which is an analogue of the Floyd–Hoare assertion system. The possibility of using these frameworks in order to build an automatic performance analyzer is discussed in a paper of Hickey and Cohen [HC88]. Their system seems to have great expressive power, but the price to be paid is the need of a complicated simplification —or even deduction— system (it is unclear whether normal forms exist in such a general calculus) and it seems hard to determine which class of programmes will be amenable to automatic treatment.

The current system originates from different traditions.

The first component —the algebraic analyzer, Alas— is based on recent research in combinatorial analysis on what can be called "symbolic methods". This approach takes its roots in works by Rota [DRS72], Foata and Schützenberger [Foa74] or Joyal [Joy81], and the reader can refer to the book by Jackson and Goulden [GJ83] for an extensive treatment of combinatorial analysis in this context. Also close in scope to our research is the work of D. Greene on labelled grammars [Gre83].

The second component —the analytic analyzer, Ananas— is rooted in a more ancient tradition of asymptotic analysis, closer to applied mathematics and analytic number theory (see e.e. [FO87] and references therein).

The interest of the $\Lambda\Upsilon^{\Omega}$ system in our view lies in the power of the "pipe" between this two components and the descriptive power that we have illustrated by examples. Some ideas on extending the symbolic approach to programme schemes originate in the work of Flajolet and Steyaert [FS81]; however the treatment given there was restricted to a class of tree algorithms, as is to some extent our first paper on the $\Lambda\Upsilon^{\Omega}$ system [FSZ88]. New impulse to these ideas has come from the contributions of Salvy and Zimmermann, which are presented in detail in companion reports [Zim88a], [Sal88].

$\Lambda\Upsilon^{\Omega}$ is not a finished product and (fortunately?), it seems to be posing more questions than it answers. We summarize here some of the problems (of varying degrees of difficulty) that we plan to explore in the

future.

1. One problem is to enlarge the class of data structures and algorithms that can be treated algebraically by the system. $\Lambda\Upsilon^{\Omega}$ is based on a certain theory of "decomposable" combinatorial structures. The translation mechanisms from programmes to complexity descriptors need more investigation so that we understand better their power and their limitations. For instance, Greene has a notion of a "boxing operator" (which fixes the location of the smallest label in a structure) which is very useful for expressing problems involving order statsitics, sorting etc. Corresponding programme constructions are yet to be found.

2. State variables ranging over a finite domain do not affect the expressive power of a programming language but they may well allow for very concise descriptions of certain programmes. (This corresponds to the old theory of Ianov programme schemes.) It seems that some benefit for the user would result from introducing such mechanisms inside $\Lambda\Upsilon^{\Omega}$.

3. In large programmes, certain components may contribute in a negligible way to the overall complexity. For instance, if we consider the scheme

```
if a=b then P(a)
       else Q(a,b) fi
```

   the resulting complexity, under a variety of conditions, will be asymptotically the same as that of an unconstrained call `Q(a,b)`, the probability of the test being satisfied being often negligible. In other words, using either probabilistic or worst–case analysis, the complexity of certain programme schemes can be qualitatively bounded and more complex, non decomposable programmes, may be ultimately analyzed. We need to understand better how such schemes can be integrated in a system.

4. The algebraic analyzer of $\Lambda\Upsilon^{\Omega}$ confronts us with a huge class of functional equations for which we need systematic asymptotic methods. At an elementary level, no one seems to have considered so far the problem of estimating the coefficients of generating functions like

$$\exp\left(\log^2 \frac{1}{1-z}\right),$$

   which falls in between known classes. More fundamentally, there is a need to unify sadlle point and singularity analysis methods. Recursive data structures pose problems related to implicit functions (selection of branches and dominant singularities) while Greene's boxing operator plunges us in the realm of singular differential systems.

In summary, $\Lambda\Upsilon^{\Omega}$ is a research enterprise whose goal is not only to assist the analysis of structurally complex programmes, but also to help individuate whole *classes* of problems whose analysis can be automated, and help discover classes of combinatorial structures and classes of "special" functions whose mathematical study should prove of interest.

Part A

Regular Languages and Finite Automata

$\Lambda\Upsilon\Omega$ Report 1

Average case analysis of a naive treatment of addition chains

I Problem specification

This simple–minded example is only meant to provide the complete $\Lambda\Upsilon\Omega$ session corresponding to the example detailed in Section 2 of the introduction. The problem is to compute an 'exponential' x^c in a group structure G (e.g. the integers modulo a prime), using the standard "binary" method.

Informally, the binary method uses the binary representation of exponent c. For instance, if $c = 29 = (11101)_2$, we have $c = 16 + 8 + 4 + 1$, and

$$x^c = x^{16} \times x^8 \times x^4 \times x,$$

which evaluates using 3 multiplications and 4 successive squaring operations.

The starting point for the analysis might be a programme in some language that implements the method which is based on the simple equations:

$$\begin{aligned} x^{2e} &= (x^e)^2 && (R0)\\ x^{2e+1} &= (x^e)^2 \cdot x\,. && (R1) \end{aligned}$$

Thus, a direct implementation in the Maple system for computer algebra is the following:

```
expmod:=proc(x,c)
# computes x^c in a group structure by squaring (.^2) and multiply (.*.)
# iquo(a,b)==a div b is the integer quotient function
        if c=0          then    1
        elif c mod 2 =0 then    c1:=iquo(c,2); (expmod(x,c1))^2
        else                    c1:=iquo(c,2); (expmod(x,c1))^2*x
        fi;
end;
```

The analysis problem consists in determining the expected number of multiplications and squaring operations that are performed in G when the exponent c is a random integer whose binary representation has length n. The corresponding Adl programme is a direct rephrasing of the Maple programme.

We find a result consistent with a direct analysis, namely that the expected number of squaring and multiply operations when the exponent has size n is

$$\tau\text{expmod}_n = \frac{3}{2}n + O(1).$$

A more complex example involving improved heuristics is given in the next report.

II Source program (Adl)

```
type chain = sequence(bit);
     bit = zero | one;
     zero,one = atom(1);

procedure expmod(c:chain);
case c of
  () : nil;
  (zero,c1) : begin expmod(c1); squaring; end;
  (one,c1)  : begin expmod(c1); squaring; multiply; end;
end;

measure multiply : 1; squaring: 1;

to_analyze : chain, expmod;
```

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
  chain(z)=Q(bit(z))
  bit(z)=zero(z)+one(z)
  zero(z)=z
  one(z)=z

Complexity descriptors:
  tau_expmod(z)=0*1+zero(z)*tau_expmod(z)/1+1*zero(z)*Q(bit(z))+0+one(z)*tau_expmod(z)/1+1*one(z)*Q
(bit(z))+1*one(z)*Q(bit(z))+0+0
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...
```

$$chain(z) = -\frac{1}{-1+2z}$$

$$tau\_expmod(z) = 3\frac{z}{(-1+2z)^2}$$

V Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...
```

```
Number of chain of size n is:
                                        infinity
                                         ---
                                         \
                                          \
                                           \
                                           /
                                          /
                                         /
                                         ---
                                          n
                                        (1/n)
                                       2    + O(n                )
                                                   (1/2)

Floating point evaluation :

                                        infinity
                                         ---
                                         \
                                          \
                                           \
                                           /
                                          /
                                         /
                                         ---
                                          n
                                        (1/n)
                                       2.   + O(n                )
                                                   (1/2)

Function expmod:
Number of inputs of expmod of size n is:
                                        infinity
                                         ---
                                         \
                                          \
                                           \
                                           /
                                          /
                                         /
                                         ---
                                          n
                                        (1/n)
                                       2    + O(n                )
                                                   (1/2)

Total cost for expmod on random inputs of size n is:
                                        (- 1 + n)           1
                                    3 2          n + O(--------)
                                                         n
                                                      (1/2)

Average cost for expmod on random inputs of size n is:

                                  av_tau_expmod_n := 3/2 n + O(1)

Floating point evaluation :

                                       1.500000000 n + O(1)
```

$\Lambda\Upsilon\Omega$ Report 2

Average case analysis of chains2

I Problem specification

This chapter analyses two heuristics for computing an exponential x^c in a group structure G. These heuristics are taken from the works of Morain and Olivos [MO89], to which we refer for a detailed presentation . They give rise to two algorithms, the second one refining the first one. In both cases, the procedure under study is `expmod`.

In both versions, procedure `expmod` implements the observation that long chains of 1's in the exponent c are better treated by division. For instance, we have

$$x^{15} = \frac{x^{16}}{x} = \frac{(((x^2)^2)^2)^2}{x},$$

which is more economical than the standard binary algorithm. In other words, *one replaces a block of at least two 1's by a block of 0's and a division.* If we imagine that we compute with exponents whose binary digits are 0, +1 and −1, then in terms of this extended representation, the algorithm corresponds to the transformation

$$1^a \mapsto 1\,0^{a-1}\text{-}1.$$

This is realised by the procedures `treat0` and `treat11`. Observe that only one division actually needs to be performed since the successive quotients can be kept in a multiplicative accumulator.

The second heuristic is a refinement of the previous rules. It uses the procedure `treat110` to extend the previous observation to isolated 0's in the binary representation of the exponent. In this case, an isolated 0 inside a block of 1's only contributes one extra division. In terms of exponent transformations, we have first by the rule of `treat11`:

$$1^a\,0\,1^b \mapsto 1\,0^{a-1}\text{-}1\,1\,0^{b-1}\,1.$$

And since $-2+1=-1$, we can pile up the transformation -1 1 ↦ -1, whence the rule describing `treat110`:

$$1^a\,0\,1^b \mapsto 1\,0^{a}\text{-}1\,0^{b-1}\text{-}1.$$

Let us denote by $\tau_n^{(1)}$ the expected cost of the first heuristic and by $\tau_n^{(2)}$ the cost of the second heuristic, cost being measured by the total number of operations performed; then $\Lambda\Upsilon\Omega$ finds that

$$\tau_n^{(1)} = \frac{11}{8}n + O(1), \qquad \tau_n^{(2)} = \frac{4}{3}n + O(1),$$

and this is to be compared to a cost of $\frac{3}{2}n$ for the naïve algorithm. Thus, we expect to save about 8.33% by the first algorithm and 11.11% by the second in terms of general arithmetic operations.

It is of interest to note that these algorithms are used when computing over elliptic curves, for exact primality testing of integers with several hundred digits. In this context, the cost of computing an inverse is essentially 0, and the cost of a squaring is slightly larger than that of a multiply. This is reflected by the

constants $k_3 = k + 3$ and $k_4 = k + 4$ in the analysis, where k is the cost of a "bignum" gcd. (The value of k is approximately 30 in this domain of application.) With this measure, we obtain, for the cost of the naïve algorithm and for the two versions of `expmod` studied here resp., the asymptotic costs

$$(\frac{1}{2}k_3 + k_4)n, \quad (\frac{3}{8}k_3 + k_4)n, \quad (\frac{1}{3}k_3 + k_4)n,$$

and for $k = 30$, the coefficients of n take the numerical values

$$50.500, \quad 46.375, \quad 45.000\,.$$

II Source program (Adl)

```
type chain = sequence(bit);
% represent an exponent with least significant bit to the left %
      valid_chain = product(chain,one);     % start with a 'one' %
      bit = zero | one;
      zero,one = atom(1);

procedure expmod(c:valid_chain);
case c of
  (c1,one) :      begin
                  expmodStart(c1); % do the job with rest of c %
                  divide           % final division %
                  end
end;

procedure expmodStart(c:chain);
case c of
  () : nil;
  (zero,c1) : begin treat0(c1) end;
  (one,c1)  : begin treat1(c1) end
end;

procedure treat0(c:chain);
case c of
  () : begin squaring; multiply end;
  (zero,c1) : begin squaring; treat0(c1) end;
  (one,c1)  : begin squaring; treat1(c1) end
end;

procedure treat1(c:chain);          % one 1 has been recognized %
case c of
  () : begin squaring; multiply; multiply end;
  (zero,c1) : begin multiply; squaring; treat0(c1) end;
  (one,c1) : begin squaring; multiply; treat11(c1) end
end;

procedure treat11(c:chain);         % at least two 1 have been recognized %
case c of
  () : begin squaring; squaring; multiply; multiply; multiply end;
  (zero,c1) : begin squaring; multiply; treat0(c1) end;
% For version 2, use instead: "(zero,c1) : begin squaring; treat110(c1) end;"%
  (one,c1)  : begin squaring; treat11(c1) end
```

```
end;

procedure treat110(c:chain); % at least two 1 and one 0 have been recognized %
case c of
  () : begin multiply; squaring; multiply end;
  (zero,c1) : begin multiply; squaring; treat0(c1) end;
  (one,c1)  : begin multiply; squaring; treat11(c1) end
end;

% For elliptic curves, we have the following costs:
   k is cost of a bignum Gcd: k =~ 30 %
measure multiply : k3;  % cost is ~ k+3 %
        divide   : k3; % cost is ~ k+3 %
        squaring : k4; % cost is ~ k+4 %

to_analyze : expmod;
```

III Algebraic analysis of the first algorithm

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
  chain(z)=Q(bit(z))
  valid_chain(z)=chain(z)*one(z)
  bit(z)=one(z)+zero(z)
  zero(z)=z
  one(z)=z

Complexity descriptors:
  tau_expmod(z)=tau_expmodStart(z)*one(z)/1+k3*chain(z)*one(z)+0
  tau_expmodStart(z)=0*1+zero(z)*tau_treat0(z)/1+one(z)*tau_treat1(z)/1+0
  tau_treat0(z)=k4*1+k3*1+k4*zero(z)*Q(bit(z))+zero(z)*tau_treat0(z)/1+k4*one(z)*Q(bit(z))+one(z)*t
au_treat1(z)/1+0
  tau_treat1(z)=k4*1+k3*1+k3*1+k3*zero(z)*Q(bit(z))+k4*zero(z)*Q(bit(z))+zero(z)*tau_treat0(z)/1+k4
*one(z)*Q(bit(z))+k3*one(z)*Q(bit(z))+one(z)*tau_treat11(z)/1+0
  tau_treat11(z)=k4*1+k4*1+k3*1+k3*1+k3*1+k4*zero(z)*Q(bit(z))+k3*zero(z)*Q(bit(z))+zero(z)*tau_tre
at0(z)/1+k4*one(z)*Q(bit(z))+one(z)*tau_treat11(z)/1+0
  tau_treat110(z)=k3*1+k4*1+k3*1+k3*zero(z)*Q(bit(z))+k4*zero(z)*Q(bit(z))+zero(z)*tau_treat0(z)/1+
k3*one(z)*Q(bit(z))+k4*one(z)*Q(bit(z))+one(z)*tau_treat11(z)/1+0
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...
```

$$\text{tau\_expmod}(z) = -\frac{z\,(3z\ k3 - 2z\ k4 - z\ k4 + 3z\ k3 - z\ k3 + 2z\ k4 - k3)}{(-1 + 2z)^2}$$

V Asymptotic analysis of the first algorithm

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Function expmod:
Number of inputs of expmod of size n is:
                                    (- 1 + n)          1
                                   2          + O(----------)
                                                      n
                                                  (1/2)  n

Total cost for expmod on random inputs of size n is:
                                       n      (- 1 + n)
                              3/16 k3 2  n + 2          k4 n + O(1)

Average cost for expmod on random inputs of size n is:

                          av_tau_expmod_n := 1/8 (3 k3 + 8 k4) n + O(1)

Floating point evaluation :

                             .1250000000 (3. k3 + 8. k4) n + O(1)
```

VI Algebraic analysis of the second algorithm

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
  chain(z)=Q(bit(z))
  valid_chain(z)=chain(z)*one(z)
  bit(z)=one(z)+zero(z)
  zero(z)=z
  one(z)=z

Complexity descriptors:
  tau_expmod(z)=tau_expmodStart(z)*one(z)/1+k3*chain(z)*one(z)+0
  tau_expmodStart(z)=0*1+zero(z)*tau_treat0(z)/1+one(z)*tau_treat1(z)/1+0
  tau_treat0(z)=k4*1+k3*1+k4*zero(z)*Q(bit(z))+zero(z)*tau_treat0(z)/1+k4*one(z)*Q(bit(z))+one(z)*t
au_treat1(z)/1+0
  tau_treat1(z)=k4*1+k3*1+k3*1+k3*zero(z)*Q(bit(z))+k4*zero(z)*Q(bit(z))+zero(z)*tau_treat0(z)/1+k4
*one(z)*Q(bit(z))+k3*one(z)*Q(bit(z))+one(z)*tau_treat11(z)/1+0
  tau_treat11(z)=k4*1+k4*1+k3*1+k3*1+k3*1+k4*zero(z)*Q(bit(z))+zero(z)*tau_treat110(z)/1+k4*one(z)*
Q(bit(z))+one(z)*tau_treat11(z)/1+0
  tau_treat110(z)=k3*1+k4*1+k3*1+k3*zero(z)*Q(bit(z))+k4*zero(z)*Q(bit(z))+zero(z)*tau_treat0(z)/1+
k3*one(z)*Q(bit(z))+k4*one(z)*Q(bit(z))+one(z)*tau_treat11(z)/1+0
```

VII Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...

                                2                     3         4       2         3
                       (- 4 z  k3 + z k3 + k3 - 4 z  k3 + 4 z  k3 + z  k4 - 4 z  k4 + 2 z k4) z
    tau_expmod(z) = - ------------------------------------------------------------------------
                                                    2
                                     (- 1 + 2 z) (2 z  + z - 1) (- 1 + z)
```

VIII Asymptotic analysis of the second algorithm

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Function expmod:
Number of inputs of expmod of size n is:
                                    (- 1 + n)          1
                                   2          + O(----------)
                                                      n
                                             (1/2)  n

Total cost for expmod on random inputs of size n is:
                                (- 1 + n)                n
                               2          k4 n + 1/6 k3 2  n + O(1)

Average cost for expmod on random inputs of size n is:

                           av_tau_expmod_n := 1/3 (3 k4 + k3) n + O(1)

Floating point evaluation :

                                .3333333333 (3. k4 + k3) n + O(1)
```

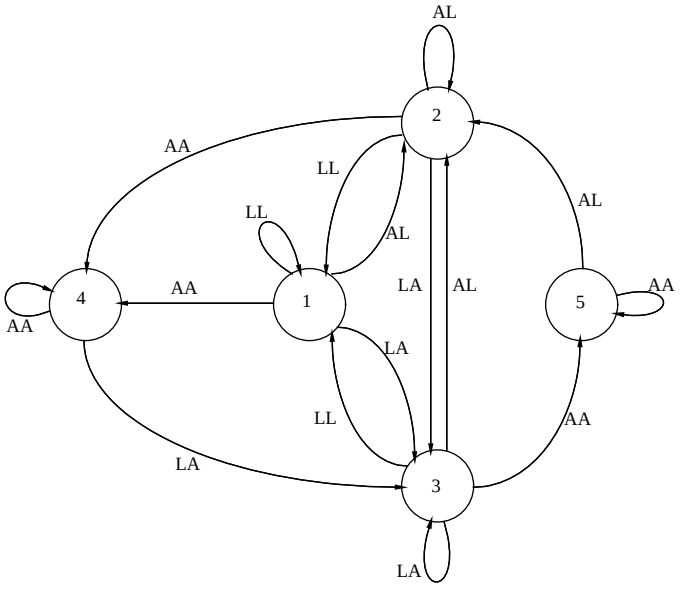
$\Lambda\Upsilon\Omega$ Report 3

Average case analysis of a concurrent access problem

I Problem specification

A simple model of the behaviour of concurrent processes can often be described in terms of finite automata. The class of problems that can be specified by regular languages and finite automata is well covered by $\Lambda\Upsilon\Omega$: Each state of the automaton is described by a corresponding procedure; each transition (labelled by a letter) is associated to a procedure to which a cost is attached. In this way, the quantitative behaviour of finite automata can be simply analyzed.

In this example we analyze a concurrent access problem along the lines of [BBT86] or [Gen89]. Two processes (P_1 and P_2) are sharing a common resource. Each process may either be in a "working phase" or in an "access phase" (where it requests the common resource), the two phases being represented by letters L ('labour') and A ('access'). Time is assumed here to be discrete so that, at any instant (time slot), each process is either in an L or A phase, and state transitions are labelled by letters from the alphabet $\{AA, AL, LA, LL\}$. In this example, there is also a "mutual exclusion" rule on the resource which can only be used by one processor at a time, and a "priority rule" by which P_1 has precedence over P_2 in case of access conflict.



State 1 means that both process are simultaneously working; state 2 means that P_1 is accessing the resource while P_2 is working, and in state 3 the roles of P_1 and P_2 are interchanged. In states 4 and 5, both processes attempt to access the resource simultaneously; however in state 4 P_1 obtains it first (the only way out is to state 3 by LA), while in state 5, P_2 obtains it first (the way out is AL). As already indicated, process P_1 has priority over process P_2: When both of them request the shared resource, and none of them used it at the preceding stage (from 1 to 4), it is always P_1 which gets it.

At the beginning, the system is in state 1. What we are looking for is the average time spent in states 4 and 5, one of the processes being waiting for the other one to finish. Here, we have also allowed different cost measures (a and b) for the time lost according to which process is waiting (P_1 or P_2 resp.).

The result which we obtain is that the first process P_1 spends about 5% of its time waiting, while this rate raises to 9% for the second process P_2, a consequence of the lower priority of P_2.

II Source program (Adl)

```
type h1 = () | product(LL,h1) | product(AL,h2) | product(AA,h4) | product(LA,h3);
     h2 = () | product(AL,h2) | product(AA,h4) | product(LL,h1) | product(LA,h3);
     h3 = () | product(LL,h1) | product(AA,h5) | product(LA,h3) | product(AL,h2);
     h4 = () | product(AA,h4) | product(LA,h3);
     h5 = () | product(AA,h5) | product(AL,h2);
     LL,AA,LA,AL = atom(1);

procedure ct1(h:h1);
case h of
  ()     : nil;
  LL(x) : ct1(x);
  AL(x) : ct2(x);
  AA(x) : begin wait2; ct4(x) end;        % The second one is waiting %
  LA(x) : ct3(x)
end;

procedure ct2(h:h2);
case h of
  ()     : nil;
  LL(x) : ct1(x);
  AL(x) : ct2(x);
  AA(x) : begin wait2; ct4(x) end;        % The second one is waiting %
  LA(x) : ct3(x)
end;

procedure ct3(h:h3);
case h of
  ()     : nil;
  LL(x) : ct1(x);
  AL(x) : ct2(x);
  AA(x) : begin wait1; ct5(x) end;        % The first one is waiting %
  LA(x) : ct3(x)
end;

procedure ct4(h:h4);
case h of
  ()     : nil;
  AA(x) : begin wait2; ct4(x) end;        % The second one is waiting %
  LA(x) : ct3(x)
```

```
end;

procedure ct5(h:h5);
case h of
  ()      : nil;
  AL(x) : ct2(x);
  AA(x) : begin wait1; ct5(x) end        % The first one is waiting %
end;

measure wait1 : a;                       % first one %
        wait2 : b;                       % second one %

to_analyze : ct1;
```

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
  h1(z)=1+LL(z)*h1(z)+AL(z)*h2(z)+AA(z)*h4(z)+LA(z)*h3(z)
  h2(z)=1+AL(z)*h2(z)+AA(z)*h4(z)+LL(z)*h1(z)+LA(z)*h3(z)
  h3(z)=1+LL(z)*h1(z)+AA(z)*h5(z)+LA(z)*h3(z)+AL(z)*h2(z)
  h4(z)=1+AA(z)*h4(z)+LA(z)*h3(z)
  h5(z)=1+AA(z)*h5(z)+AL(z)*h2(z)
  LL(z)=z
  AA(z)=z
  LA(z)=z
  AL(z)=z

Complexity descriptors:
  tau_ct1(z)=0*1+LL(z)*tau_ct1(z)/1+AL(z)*tau_ct2(z)/1+b*AA(z)*h4(z)+AA(z)*tau_ct4(z)/1+LA(z)*tau_c
t3(z)/1+0
  tau_ct2(z)=0*1+LL(z)*tau_ct1(z)/1+AL(z)*tau_ct2(z)/1+b*AA(z)*h4(z)+AA(z)*tau_ct4(z)/1+LA(z)*tau_c
t3(z)/1+0
  tau_ct3(z)=0*1+LL(z)*tau_ct1(z)/1+AL(z)*tau_ct2(z)/1+a*AA(z)*h5(z)+AA(z)*tau_ct5(z)/1+LA(z)*tau_c
t3(z)/1+0
  tau_ct4(z)=0*1+b*AA(z)*h4(z)+AA(z)*tau_ct4(z)/1+LA(z)*tau_ct3(z)/1+0
  tau_ct5(z)=0*1+AL(z)*tau_ct2(z)/1+a*AA(z)*h5(z)+AA(z)*tau_ct5(z)/1+0
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...
```

$$\tau_{ct1}(z) = -\frac{z\,(2z-1)\,(-2bz+z^2b+za+b)}{(2z^2-4z+1)\,(-6z^3-5z^2+7z+2z^4+1)}$$

V Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Function ct1:
Number of inputs of ct1 of size n is:
                        1/2     (- n)  (1/2 n - 1/2)
                     (2    - 1)      2                          1
                  - ------------------------------ + O(--------------------)
                                    1/2                          1/2 n
                               - 2 + 2                   (1 - 1/2 2    )  n

Total cost for ct1 on random inputs of size n is:
                             1                                      1
    b (- 1/2 ---------------------------------- + 1/2 ----------------------------)
                      1/2            1/2   1/2                1/2            1/2
             (2 - 3 2   ) (1 - 1/2 2   ) 2            (2 - 3 2   ) (1 - 1/2 2   )
(- ------------------------------------------------------------------------------
                                          1/2 n
                                   (1 - 1/2 2   )

                        1                       1
      a (- 1/2 ------------------- + 1/2 ------------)
                      1/2   1/2                 1/2
              (2 - 3 2   ) 2            2 - 3 2
   - ------------------------------------------------
                                 1/2 n
                     (1 - 1/2 2   )

                                                          1/2
                        1                       1        2
      b (- 1/2 ------------------- + 3/4 ------------ - 1/4 ------------)
                      1/2   1/2                 1/2              1/2
              (2 - 3 2   ) 2            2 - 3 2          2 - 3 2
   - ------------------------------------------------------------------
                                    1/2 n
                           (1 - 1/2 2   )

                          1                       1
        b (- 1/2 ------------------- + 1/2 ------------)
                        1/2   1/2                 1/2
                (2 - 3 2   ) 2            2 - 3 2
   + 2 ------------------------------------------------)
                                  1/2 n
                         (1 - 1/2 2   )

   n

   + O(1)

Average cost for ct1 on random inputs of size n is:

av_tau_ct1_n :=

   2

                                   1                                      1
      (- b (- 1/2 ---------------------------------- + 1/2 ----------------------------)
                        1/2            1/2   1/2                1/2            1/2
                (2 - 3 2   ) (1 - 1/2 2   ) 2            (2 - 3 2   ) (1 - 1/2 2   )
```

```
                          1                          1
      - a (- 1/2 -------------------- + 1/2 ------------)
                        1/2   1/2                 1/2
                (2 - 3 2    ) 2             2 - 3 2

                                                                 1/2
                          1                          1          2
      - b (- 1/2 -------------------- + 3/4 ------------ - 1/4 ------------)
                        1/2   1/2                 1/2                1/2
                (2 - 3 2    ) 2             2 - 3 2            2 - 3 2

                            1                          1
      + 2 b (- 1/2 -------------------- + 1/2 ------------))
                          1/2   1/2                 1/2
                  (2 - 3 2    ) 2             2 - 3 2

       1/2            1/2
      2    (1 - 1/2 2    ) n

    + O(1)
```

Floating point evaluation :

$$.8284271250\ (.1114757265\ b + .0653009687\ a)\ n + O(1)$$

ΛγΩ Report 4

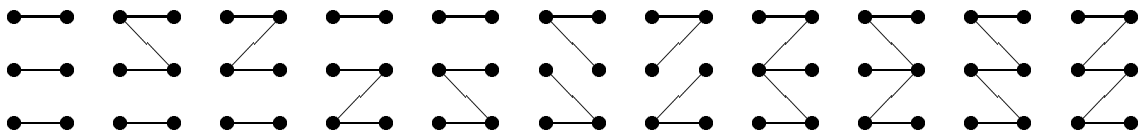
Average case analysis of a combinatorial problem (1)

I Problem specification

This problem was introduced by Steven Bird in the transaction `<1107@epistemi.ed.ac.uk>` of the newsgroup *Sci.math* (see Report 18 for a similar problem):

> `Given two rows consisting of n points each, how many ways are there of drawing straight lines between them so that (i) from each point there is a line, (ii) no lines cross, and (iii) a line from point i in row a can only go to points i-1,i,i+1 in row b, and vice versa. This function, h(n), has the following values for small n: 1, 3, 11, 41, 153, 571, 2131, 7953, 29681.`

The 11 possible configurations for $n = 3$ are:



For a point i of the left row (a), there are 3 possible lines towards points of row b: one to point $i-1$, one to point i and one to point $i+1$. We will denote these lines `p`, `c` and `f` for "previous", "current" and "following". So we have a *priori* $2^3 = 8$ possible configurations for each point of leftmost row, since connections of a leftmost point can be described by one of the 8 symbols $\{\epsilon, p, c, f, pc, cf, fp, pcf\}$. For example, the 11 configurations shown above are encoded by

$$(c,c,c)\ (cf,c,c)\ (c,pc,c)\ (c,c,pc)\ (c,cf,c)\ (cf,f,c)\ (c,p,pc)\ (c,pcf,c)\ (cf,c,pc)\ (cf,cf,c)\ (c,pc,pc)$$

The empty configuration is forbidden (condition (i) for i in row a), and also `pf` (other lines could not cross `p` and `f`, so point i in row b would remain isolated). Thus, the possible choices for a point in row a are described by the set $\mathcal{A} = \{\mathtt{p}, \mathtt{c}, \mathtt{f}, \mathtt{pc}, \mathtt{cf}, \mathtt{pcf}\}$. Hence a solution for n points in each row is a word of length n over the 6-letter alphabet $\mathcal{A}$. However, there are additional conditions for a valid configuration (we can not start with `p`, `pc` or `pcf`, we must end with `c` or `pc`), and some pairs of letters are forbidden by condition (i) [an $\mathcal{A}$-letter containing `p` cannot precede an $\mathcal{A}$-letter containing `f`] or by condition (ii) [an $\mathcal{A}$-letter containing `f` cannot precede an $\mathcal{A}$-letter containing `p`].

The procedure `valid_start` takes as input a word over this 6-letter alphabet, and ends with `yes` if this word corresponds to a valid configuration, with `no` otherwise. Hence, if the cost of `yes` is set to 1 (and the cost of `no` to 0), the complexity descriptor of `valid_start` is exactly the counting generating function for valid configurations.

We deduce the answer to the original problem: the number of ways of drawing straight lines between two rows of n points so that (i), (ii) and (iii) are satisfied is the coefficient of z^n in:

$$\frac{1-3z}{1-4z+z^2}$$

which satisfies asymptotically:

$$h(n) = CA^n + O(A^n/n)$$

with $A = 2+\sqrt{3} \simeq 3.732050808$ and $C = (1-1/\sqrt{3})/2 \simeq 0.2113248655$.

II Source program (Adl)

```
type choice = p | c | f                    % previous, current, following %
            | pc | cf | pcf;              % p and c, c and f, p and c and f %
     graph = sequence(choice);
     p, c, f, pc, cf, pcf = atom(1);

procedure valid_start(g : graph);
case g of                                  % we can only start with c or cf %
  ()         : yes;
  (c,h)      : valid_after_c(h);
  (cf,h)     : valid_after_f(h);
  otherwise : no;
end;

procedure valid_after_p(g : graph);
case g of
  ()         : no;                         % the last point remains %
  (p,h)      : valid_after_p(h);
  (c,h)      : no;                         % the above point remains %
  (f,h)      : no;                         % above and current point remain %
  (pc,h)     : valid_after_c(h);
  (cf,h)     : no;                         % the above point remains %
  (pcf,h)    : valid_after_f(h);
end;

procedure valid_after_c(g : graph);        % p, c, pc, cf, pcf are allowed %
case g of
  ()         : yes;
  (p,h)      : valid_after_p(h);
  (c,h)      : valid_after_c(h);
  (f,h)      : no;                         % the current point will remain %
  (pc,h)     : valid_after_c(h);
  (cf,h)     : valid_after_f(h);
  (pcf,h)    : valid_after_f(h);
end;

procedure valid_after_f(g : graph);
case g of
  ()         : no;                         % there must be a point below %
  (p,h)      : no;                         % lines f(i-1) and p(i) cross %
  (c,h)      : valid_after_c(h);
  (f,h)      : valid_after_f(h);
```

```
    (pc,h)     : no;                          % lines f(i−1) and p(i) cross %
    (cf,h)     : valid_after_f(h);
    (pcf,h)    : no;                          % lines f(i−1) and p(i) cross %
  end;

  measure yes : 1; no : 0;

  to_analyze : valid_start;
```

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
  choice(z)=p(z)+c(z)+f(z)+pc(z)+cf(z)+pcf(z)
  graph(z)=Q(choice(z))
  p(z)=z
  c(z)=z
  f(z)=z
  pc(z)=z
  cf(z)=z
  pcf(z)=z

Complexity descriptors:
  tau_valid_start(z)=1*1+c(z)*tau_valid_after_c(z)/1+cf(z)*tau_valid_after_f(z)/1+0*(p(z)+f(z)+pc(z
)+pcf(z))*Q(choice(z))
  tau_valid_after_p(z)=0*1+p(z)*tau_valid_after_p(z)/1+0*c(z)*Q(choice(z))+0*f(z)*Q(choice(z))+pc(z
)*tau_valid_after_c(z)/1+0*cf(z)*Q(choice(z))+pcf(z)*tau_valid_after_f(z)/1+0
  tau_valid_after_c(z)=1*1+p(z)*tau_valid_after_p(z)/1+c(z)*tau_valid_after_c(z)/1+0*f(z)*Q(choice(
z))+pc(z)*tau_valid_after_c(z)/1+cf(z)*tau_valid_after_f(z)/1+pcf(z)*tau_valid_after_f(z)/1+0
  tau_valid_after_f(z)=0*1+0*p(z)*Q(choice(z))+c(z)*tau_valid_after_c(z)/1+f(z)*tau_valid_after_f(z
)/1+0*pc(z)*Q(choice(z))+cf(z)*tau_valid_after_f(z)/1+0*pcf(z)*Q(choice(z))+0
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...
```

$$\text{tau\_valid\_start}(z) = -\frac{3z-1}{z^2-4z+1}$$

V Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Function valid_start:
Number of inputs of valid_start of size n is:
                                                  infinity
```

```
                                    n      (1/n)
                                   6  + O(---------------)
                                                  n
                                            (1/6)
```

Total cost for valid_start on random inputs of size n is:

```
                                 3/2        1/2 (- n)
                          (5 - 3   ) (2 - 3   )                  1
                    1/2 ---------------------------- + O(---------------)
                                 1/2         1/2                  1/2 n
                                3    (- 2 + 3   )          (2 - 3   )  n
```

Average cost for valid_start on random inputs of size n is:

```
                                      3/2   (- n)       1/2 (- n)             1/2 (- n)      n
                                (5 - 3   ) 6      (2 - 3   )            (2 - 3   )      (1/6)
   av_tau_valid_start_n := 1/2 ---------------------------------- + O(------------------------)
                                     1/2         1/2                           n
                                    3    (- 2 + 3   )
```

Floating point evaluation :

```
                                                                1/2 (- n)      n
                          (- 1. n)             (- 1. n)      (2 - 3   )      (1/6)
             .2113248692 6.         (.267949192)         + O(------------------------)
                                                                      n
```

Part B

Context–Free Languages, Terms and Symbolic Manipulation Algorithms

$\Lambda\Upsilon\Omega$ Report 5

Average case analysis of differentiation algorithms

I Problem specification

We examine here two versions of the differentiation algorithm, `diff` and `diff1`. Both apply to expression trees generated from constants 0, 1, from variable x and from function symbols $+$, $\times$, and exp(.). The differentiation rules are the usual ones. Letting D denote derivatives with respect to x, we have:

$$\begin{cases} D0 = 0 & D1 = 0 & Dx = 1 \\ D(f+g) = Df + Dg, & D(f \cdot g) = f \cdot Dg + Df \cdot g & D(e^f) = e^f \cdot Df. \end{cases}$$

The algorithm `diff` is the one we considered already in Section 1 of the introduction to this paper. It shares its subtrees with its argument. For instance, when applying the rule for differentiating a product $f \cdot g$ we do not copy f or g but rather "share" them with the original expression. In this way, we only traverse a tree, and occasionally generate a bounded number of symbols. Thus the worst case complexity is of order $O(n)$, as is the average case complexity.

The algorithm `diff1` operates by copying subtrees instead of sharing them. In performing differentiation with this algorithm, we thus obtain trees that tend to be larger than the original trees. The growth can be precisely quantified, and it turns out that differentiation with copy has an average cost of $O(n^{3/2})$.

$\Lambda\Upsilon\Omega$ assigns a constant cost to procedure calls, which corresponds to a call by name mechanism. This example illustrates the possiblity of analyzing calls by value through the use of copy procedures. The next report gives the analysis of repeated differentiation.

II Source program (Adl)

```
type expression = zero | one | x
                | product(plus,expression,expression)
                | product(times,expression,expression)
                | product(expo,expression);
plus,times,expo,zero,one,x = atom(1);

% Differentiation algorithm with sharing of subexpressions %
function diff(e:expression):expression;
case e of
  plus(e1,e2)     : plus(diff(e1),diff(e2));
  times(e1,e2)    : plus(times(diff(e1),e2),
```

```
                        times(e1,diff(e2)));
  expo(e1)      : times(diff(e1),e);
  zero          : zero;
  one           : zero;
  x             : one
end;

% Differentiation algorithm with copy of subexpressions %
function diff1(e:expression):expression;
case e of
  plus(e1,e2)   : plus(diff1(e1),diff1(e2));
  times(e1,e2)  : plus(times(diff1(e1),copy(e2)),
                        times(copy(e1),diff1(e2)));
  expo(e1)      : times(diff1(e1),copy(e));
  zero          : zero;
  one           : zero;
  x             : one
end;

function copy (e:expression):expression;
case e of
  plus(e1,e2)   : plus(copy(e1),copy(e2));
  times(e1,e2)  : times(copy(e1),copy(e2));
  expo(e1)      : expo(copy(e1));
  zero          : zero;
  one           : one;
  x             : x
end;

measure plus,times,expo,zero,one,x : 1;
        % the cost of a function is the number of nodes generated %

to_analyze : diff,diff1;
```

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
  expression(z)=expo(z)*expression(z)+times(z)*expression(z)*expression(z)+plus(z)*expression(z)*ex
pression(z)+x(z)+one(z)+zero(z)
  plus(z)=z
  times(z)=z
  expo(z)=z
  zero(z)=z
  one(z)=z
  x(z)=z

Complexity descriptors:
  tau_diff(z)=0*plus(z)*expression(z)*expression(z)+1*plus(z)*expression(z)*expression(z)+plus(z)*t
au_diff(z)*expression(z)/1+plus(z)*expression(z)*tau_diff(z)/1+0*times(z)*expression(z)*expression(
z)+1*times(z)*expression(z)*expression(z)+0*times(z)*expression(z)*expression(z)+1*times(z)*express
ion(z)*expression(z)+times(z)*tau_diff(z)*expression(z)/1+0*times(z)*expression(z)*expression(z)+0*
```

```
times(z)*expression(z)*expression(z)+1*times(z)*expression(z)*expression(z)+0*times(z)*expression(z
)*expression(z)+times(z)*expression(z)*tau_diff(z)/1+0*expo(z)*expression(z)+1*expo(z)*expression(z
)+expo(z)*tau_diff(z)/1+0*expo(z)*expression(z)+1*zero(z)+1*one(z)+1*x(z)+0
  tau_diff1(z)=0*plus(z)*expression(z)*expression(z)+1*plus(z)*expression(z)*expression(z)+plus(z)*
tau_diff1(z)*expression(z)/1+plus(z)*expression(z)*tau_diff1(z)/1+0*times(z)*expression(z)*expressi
on(z)+1*times(z)*expression(z)*expression(z)+0*times(z)*expression(z)*expression(z)+1*times(z)*expr
ession(z)*expression(z)+times(z)*tau_diff1(z)*expression(z)/1+times(z)*expression(z)*tau_copy(z)/1+
0*times(z)*expression(z)*expression(z)+1*times(z)*expression(z)*expression(z)+times(z)*tau_copy(z)*
expression(z)/1+times(z)*expression(z)*tau_diff1(z)/1+0*expo(z)*expression(z)+1*expo(z)*expression(
z)+expo(z)*tau_diff1(z)/1+1*(0*expo(z)*expression(z)+1*expo(z)*expression(z)+expo(z)*tau_copy(z)/1+
0)+1*zero(z)+1*one(z)+1*x(z)+0
  tau_copy(z)=0*plus(z)*expression(z)*expression(z)+1*plus(z)*expression(z)*expression(z)+plus(z)*t
au_copy(z)*expression(z)/1+plus(z)*expression(z)*tau_copy(z)/1+0*times(z)*expression(z)*expression(
z)+1*times(z)*expression(z)*expression(z)+times(z)*tau_copy(z)*expression(z)/1+times(z)*expression(
z)*tau_copy(z)/1+0*expo(z)*expression(z)+1*expo(z)*expression(z)+expo(z)*tau_copy(z)/1+1*zero(z)+1*
one(z)+1*x(z)+0
  tau_copy1(z)=0*expo(z)*expression(z)+1*expo(z)*expression(z)+expo(z)*tau_copy(z)/1+0
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...

tau_diff(z)
```

$$= -\left(-\frac{1}{4}\frac{1}{z} + \frac{1}{4} + \frac{1}{2}\frac{\sqrt{(1 - 2\,z - 23\,z^2)}}{z} - 3\,z - \frac{1}{4}\sqrt{(1 - 2\,z - 23\,z^2)}\right)\Big/\left(-\frac{1}{4}\frac{1 - 2\,z - 23\,z^2}{z}\right)\Big/\sqrt{(1 - 2\,z - 23\,z^2)}$$

```
tau_diff1(z)
```

$$= -\left(\frac{3}{8}\frac{\sqrt{(1 - 2\,z - 23\,z^2)}}{z} - \frac{11}{4}\,z - \frac{1}{4}\frac{1 - 2\,z - 23\,z^2}{z} - \frac{1}{8}\frac{1}{z\sqrt{(1 - 2\,z - 23\,z^2)}}\right)\Big/\left(\frac{1}{8}\frac{z}{\sqrt{(1 - 2\,z - 23\,z^2)}}\right)\Big/\sqrt{(1 - 2\,z - 23\,z^2)}$$

V Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Function diff:
Number of inputs of diff of size n is:
                                 1/2 (- n)   1/2  1/4
                    (- 1/23 + 2/23 6    )      23    6                     1
               1/4 ------------------------------------ + O(--------------------------)
                                1/2 1/2  3/2   1/2                          1/2 n  2
                       (- 1 + 2 6   )    n    Pi              (- 1/23 + 2/23 6   )  n

Total cost for diff on random inputs of size n is:
                                                                        1/4
  213                   1                        47                 6
 ----- ---------------------------------- - ------ -----------------------------
  529    1/2  1/4                 1/2 3/2     1058    1/2                 1/2 3/2
        4    6    (- 1/23 + 2/23 6   )               4    (- 1/23 + 2/23 6   )
-------------------------------------------------------------------------------
                                       1/2 n  1/2   1/2
                          (- 1/23 + 2/23 6   )  n    Pi

                     1
   + O(----------------------------)
                         1/2 n  3/2
         (- 1/23 + 2/23 6   )  n

Average cost for diff on random inputs of size n is:

av_tau_diff_n :=

   8

                                                                      1/4
       213                   1                        47                 6
      (----- ---------------------------------- - ------ -----------------------------)
       529    1/2  1/4                 1/2 3/2     1058    1/2                 1/2 3/2
             4    6    (- 1/23 + 2/23 6   )               4    (- 1/23 + 2/23 6   )

                         1/2 1/2
        (- 1/23 + 2/23 6   )    n

            1/2  1/4
        / (4    6   )

           1/2
      + O(n   )

Floating point evaluation :

                                                  1/2
                                  1.415239577 n + O(n   )

Function diff1:
Number of inputs of diff1 of size n is:
                                 1/2 (- n)   1/2  1/4
                    (- 1/23 + 2/23 6    )      23    6                     1
               1/4 ------------------------------------ + O(--------------------------)
                                1/2 1/2  3/2   1/2                          1/2 n  2
                       (- 1 + 2 6   )    n    Pi              (- 1/23 + 2/23 6   )  n
```

```
Total cost for diff1 on random inputs of size n is:
   63                1                                  1
 ------ ---------------------------- + 1/4232 -----------------------
  2116                   1/2 2  1/2                          1/2 2
         (- 1/23 + 2/23 6   )  6                (- 1/23 + 2/23 6   )
-------------------------------------------------------------------------
                                      1/2 n
                          (- 1/23 + 2/23 6   )

              1/2                  1/2 1/2                        1/2
             4    (- 1/23 + 2/23 6   )                          4
   + (- 3/4 ------------------------------ + 1/16 -------------------------------
                           1/4                          1/4                  1/2 3/2
                          6                            6    (- 1/23 + 2/23 6   )

                           1/2
                          4
      - 1/8 ------------------------------)
              1/4                  1/2 1/2
             6    (- 1/23 + 2/23 6   )

                            1/2 n  1/2   1/2
     1 / ((- 1/23 + 2/23 6   )  n    Pi   )

                  1
   + O(-------------------------)
                         1/2 n
        (- 1/23 + 2/23 6   )  n

Average cost for diff1 on random inputs of size n is:

                                                    1/2    1/2  3/2
                                              (126 + 6   ) Pi    n
                  av_tau_diff1_n := 1/2 ------------------------------ + O(n)
                                                  1/2 3/2  3/4   1/2
                                          (- 1 + 2 6   )    6    23

Floating point evaluation :

                                              3/2
                                   .8042175435 n    + O(n)
```

$\Lambda\Upsilon\Omega$ Report 6

Average case analysis of Derivatives of Higher Order

I Problem specification

This chapter gives an analysis of iterated differentiation. As explained at the end of section III.2 of the introduction, $\Lambda\Upsilon\Omega$ does not allow functional composition. In other words, the result of a user–defined procedure cannot be taken as input of another procedure since there is no variable assignment mechanism. The example given here shows some cases where this limitation can be bypassed.

The principle used to analyze iterated differentiation consists simply in setting up explicit procedures that perform directly repeated differentiation. For instance, the third derivative of the exponential function will be computed by:

$$\frac{d^3}{dx^3}\exp(f(x)) = (f^{(3)}(x) + 3f''(x)f'(x) + f'(x)^3)\exp(f(x)).$$

(In this case, the output expression will involve an extended set of operators like `three` meaning '$3 \cdot (.)$' or `sqr3` meaning '$(.)^3$'. These operators are all asigned unit cost; it can be checked, again with $\Lambda\Upsilon\Omega$, that such simplifications do not affect the growth order of complexities.)

We saw in the last chapter that a single differentiation has cost $O(n^{3/2})$. This suggests that a k-th order derivative might have cost $O(n^{(3/2)^k})$. The analysis given here illustrates the perhaps counter–intuitive fact that the complexity of a k–th order derivative is only $O(n^{k/2+1})$.

II Source program (Adl)

```
type expression = zero | one | x
                | plus(expression,expression)
                | times(expression,expression)
                | expo(expression);
     plus,times,expo,zero,one,x = atom(1);

function diff(e:expression):expression;
case e of
  plus(e1,e2)     : plus(diff(e1),diff(e2));
  times(e1,e2)    : plus(times(diff(e1),copy(e2)),times(copy(e1),diff(e2)));
  expo(e1)        : times(diff(e1),copy(e));
  zero            : zero;
  one             : zero;
  x               : one
```

```
end;

function copy(e:expression):expression;
case e of
  plus(e1,e2)     : plus(copy(e1),copy(e2));
  times(e1,e2)    : times(copy(e1),copy(e2));
  expo(e1)        : expo(copy(e1));
  zero            : zero;
  one             : one;
  x               : x
end;

function diff2(e:expression):expression;
case e of
  plus(t,u)       : plus(diff2(t),diff2(u));
  times(t,u)      : plus(times(diff2(t),copy(u)),                 % t" u %
                         times(two,diff(t),diff(u)),              % 2 t' u'%
                         times(copy(t),diff2(u)));                % t u" %
  expo(t)         : times(plus(diff2(t),sqr(diff(t))),
                          copy(e));                               % (t"+t'^2) e %
  otherwise       : zero
end;

function diff3(e:expression):expression;
case e of
  plus(t,u)       : plus(diff3(t),diff3(u));
  times(t,u)      : plus(times(diff3(t),copy(u)),
                         times(three,diff2(t),diff(u)),
                         times(three,diff(t),diff2(u)),
                         times(copy(t),diff3(u)));
  expo(t)         : times(plus(diff3(t),
                               times(three,diff2(t),diff(t)),
                               sqr3(diff(t))),
                          copy(e));
  otherwise       : zero
end;

measure plus,times,expo,zero,one,x,three,two,sqr,sqr3 : 1;

to_analyze : diff2,diff3;
```

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
  expression(z)=zero(z)+one(z)+x(z)+plus(z)*expression(z)*expression(z)+times(z)*expression(z)*expr
ession(z)+expo(z)*expression(z)
  plus(z)=z
  times(z)=z
  expo(z)=z
  zero(z)=z
  one(z)=z
```

```
  x(z)=z

Complexity descriptors:
  tau_diff(z)=0*plus(z)*expression(z)*expression(z)+1*plus(z)*expression(z)*expression(z)+plus(z)*t
au_diff(z)*expression(z)/1+plus(z)*expression(z)*tau_diff(z)/1+0*times(z)*expression(z)*expression(
z)+1*times(z)*expression(z)*expression(z)+0*times(z)*expression(z)*expression(z)+1*times(z)*express
ion(z)*expression(z)+times(z)*tau_diff(z)*expression(z)/1+times(z)*expression(z)*tau_copy(z)/1+0*ti
mes(z)*expression(z)*expression(z)+1*times(z)*expression(z)*expression(z)+times(z)*tau_copy(z)*expr
ession(z)/1+times(z)*expression(z)*tau_diff(z)/1+0*expo(z)*expression(z)+1*expo(z)*expression(z)+ex
po(z)*tau_diff(z)/1+1*(0*expo(z)*expression(z)+1*expo(z)*expression(z)+expo(z)*tau_copy(z)/1+0)+1*z
ero(z)+1*one(z)+1*x(z)+0
  tau_copy(z)=0*plus(z)*expression(z)*expression(z)+1*plus(z)*expression(z)*expression(z)+plus(z)*t
au_copy(z)*expression(z)/1+plus(z)*expression(z)*tau_copy(z)/1+0*times(z)*expression(z)*expression(
z)+1*times(z)*expression(z)*expression(z)+times(z)*tau_copy(z)*expression(z)/1+times(z)*expression(
z)*tau_copy(z)/1+0*expo(z)*expression(z)+1*expo(z)*expression(z)+expo(z)*tau_copy(z)/1+1*zero(z)+1*
one(z)+1*x(z)+0
  tau_diff2(z)=0*plus(z)*expression(z)*expression(z)+1*plus(z)*expression(z)*expression(z)+plus(z)*
tau_diff2(z)*expression(z)/1+plus(z)*expression(z)*tau_diff2(z)/1+0*times(z)*expression(z)*expressi
on(z)+1*times(z)*expression(z)*expression(z)+0*times(z)*expression(z)*expression(z)+1*times(z)*expr
ession(z)*expression(z)+times(z)*tau_diff2(z)*expression(z)/1+times(z)*expression(z)*tau_copy(z)/1+
0*times(z)*expression(z)*expression(z)+1*times(z)*expression(z)*expression(z)+1*times(z)*expression
(z)*expression(z)+times(z)*tau_diff(z)*expression(z)/1+times(z)*expression(z)*tau_diff(z)/1+0*times
(z)*expression(z)*expression(z)+1*times(z)*expression(z)*expression(z)+times(z)*tau_copy(z)*express
ion(z)/1+times(z)*expression(z)*tau_diff2(z)/1+0*expo(z)*expression(z)+1*expo(z)*expression(z)+0*ex
po(z)*expression(z)+1*expo(z)*expression(z)+expo(z)*tau_diff2(z)/1+1*expo(z)*expression(z)+expo(z)*
tau_diff(z)/1+1*(0*expo(z)*expression(z)+1*expo(z)*expression(z)+expo(z)*tau_copy(z)/1+0)+1*(zero(z
)+one(z)+x(z))
  tau_diff3(z)=0*plus(z)*expression(z)*expression(z)+1*plus(z)*expression(z)*expression(z)+plus(z)*
tau_diff3(z)*expression(z)/1+plus(z)*expression(z)*tau_diff3(z)/1+0*times(z)*expression(z)*expressi
on(z)+1*times(z)*expression(z)*expression(z)+0*times(z)*expression(z)*expression(z)+1*times(z)*expr
ession(z)*expression(z)+times(z)*tau_diff3(z)*expression(z)/1+times(z)*expression(z)*tau_copy(z)/1+
0*times(z)*expression(z)*expression(z)+1*times(z)*expression(z)*expression(z)+1*times(z)*expression
(z)*expression(z)+times(z)*tau_diff2(z)*expression(z)/1+times(z)*expression(z)*tau_diff(z)/1+0*time
s(z)*expression(z)*expression(z)+1*times(z)*expression(z)*expression(z)+1*times(z)*expression(z)*ex
pression(z)+times(z)*tau_diff(z)*expression(z)/1+times(z)*expression(z)*tau_diff2(z)/1+0*times(z)*e
xpression(z)*expression(z)+1*times(z)*expression(z)*expression(z)+times(z)*tau_copy(z)*expression(z
)/1+times(z)*expression(z)*tau_diff3(z)/1+0*expo(z)*expression(z)+1*expo(z)*expression(z)+0*expo(z)
*expression(z)+1*expo(z)*expression(z)+expo(z)*tau_diff3(z)/1+0*expo(z)*expression(z)+1*expo(z)*exp
ression(z)+1*expo(z)*expression(z)+expo(z)*tau_diff2(z)/1+expo(z)*tau_diff(z)/1+1*expo(z)*expressio
n(z)+expo(z)*tau_diff(z)/1+1*(0*expo(z)*expression(z)+1*expo(z)*expression(z)+expo(z)*tau_copy(z)/1
+0)+1*(zero(z)+one(z)+x(z))
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...

tau_diff2(z)
```

$$= -\left(-1 - 2z - 6\,\frac{z^2}{(1 - 2z - 23z^2)^{2\,1/2}} - 3\,\frac{z^3}{1 - 2z - 23z^2}\right)$$

```
                                  2 1/2 2                                  2 1/2 2
         (- 1 + z + (1 - 2 z - 23 z )   )         (- 1 + z + (1 - 2 z - 23 z )   )
    - 3/4 ---------------------------------- - 3/8 ----------------------------------
                               2 1/2                                z
                (1 - 2 z - 23 z )

                                     2 1/2                                    2 1/2 4
          z (- 1 + z + (1 - 2 z - 23 z )   )         (- 1 + z + (1 - 2 z - 23 z )   )
    + 15/4 ----------------------------------- - 1/32 ----------------------------------
                               2 1/2                                    2
                 (1 - 2 z - 23 z )                        z (1 - 2 z - 23 z )

                                                    2 1/2 3
                   2 1/2         (- 1 + z + (1 - 2 z - 23 z )   )
    + (1 - 2 z - 23 z )    + 3/16 ----------------------------------
                                                  2 1/2
                                     z (1 - 2 z - 23 z )

           2                           2 1/2                                    2 1/2 2
          z  (- 1 + z + (1 - 2 z - 23 z )   )         z (- 1 + z + (1 - 2 z - 23 z )   )
    + 13/4 ----------------------------------- - 9/8 ----------------------------------
                                  2                                        2
                   1 - 2 z - 23 z                           1 - 2 z - 23 z

                                    2 1/2 3
          (- 1 + z + (1 - 2 z - 23 z )   )
    + 3/16 ----------------------------------)
                                  2
                   1 - 2 z - 23 z

                   2 1/2
    / (1 - 2 z - 23 z )
```

tau_diff3(z)

```
                                2                          3
                               z                          z
    = - (- 3/2 z - 12 ---------------------- - 12 -----------------
                                  2 1/2                        2
                   (1 - 2 z - 23 z )          1 - 2 z - 23 z

                                     2 1/2 2                                  2 1/2 2
          (- 1 + z + (1 - 2 z - 23 z )   )         (- 1 + z + (1 - 2 z - 23 z )   )
    - 15/8 ----------------------------------- - 1/2 ----------------------------------
                               2 1/2                                z
                 (1 - 2 z - 23 z )

                                     2 1/2                                    2 1/2 4
          z (- 1 + z + (1 - 2 z - 23 z )   )         (- 1 + z + (1 - 2 z - 23 z )   )
    + 27/4 ----------------------------------- - 1/8 ----------------------------------
                               2 1/2                                    2
                 (1 - 2 z - 23 z )                        z (1 - 2 z - 23 z )

                                                    2 1/2 3
                     2 1/2         (- 1 + z + (1 - 2 z - 23 z )   )
    + 3/2 (1 - 2 z - 23 z )    + 3/8 ----------------------------------
                                                  2 1/2
                                     z (1 - 2 z - 23 z )
```

```
          2                          2 1/2                  4
         z  (- 1 + z + (1 - 2 z - 23 z )   )              z
+ 47/4 ------------------------------------ - 3 ---------------------- - 3/2
                                2                            2 3/2
                 1 - 2 z - 23 z                 (1 - 2 z - 23 z )

                                 2 1/2 2                                  2 1/2 3
     z (- 1 + z + (1 - 2 z - 23 z )   )      13   (- 1 + z + (1 - 2 z - 23 z )   )
- 4 ------------------------------------ + ---- ---------------------------------
                                2              16                        2
              1 - 2 z - 23 z                             1 - 2 z - 23 z

          3                          2 1/2                                  2 1/2 5
         z  (- 1 + z + (1 - 2 z - 23 z )   )          (- 1 + z + (1 - 2 z - 23 z )   )
+ 19/4 ------------------------------------ + 1/64 ---------------------------------
                                2 3/2                                  2 3/2
               (1 - 2 z - 23 z )                              z (1 - 2 z - 23 z )

                                   2 1/2 3           2                          2 1/2 2
       z (- 1 + z + (1 - 2 z - 23 z )   )          z  (- 1 + z + (1 - 2 z - 23 z )   )
+ 3/4 ------------------------------------ - 11/4 ------------------------------------
                                2 3/2                                  2 3/2
              (1 - 2 z - 23 z )                              (1 - 2 z - 23 z )

                                   2 1/2 4
       (- 1 + z + (1 - 2 z - 23 z )   )
- 1/8 ----------------------------------)
                                2 3/2
             (1 - 2 z - 23 z )

                 2 1/2
/ (1 - 2 z - 23 z )
```

V Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Function diff2:
Number of inputs of diff2 of size n is:
                                1/2 (- n)   1/2  1/4
                    (- 1/23 + 2/23 6   )      23    6                     1
               1/4 ------------------------------------ + O(--------------------------)
                                1/2 1/2  3/2   1/2                          1/2 n  2
                     (- 1 + 2 6   )    n    Pi              (- 1/23 + 2/23 6   )  n

Total cost for diff2 on random inputs of size n is:
2

                                                                 1/2  1/4
     115                   1/2 3/2  1/2  1/4     125           4    6
   (------ (- 1/23 + 2/23 6   )    4    6    - ------- -------------------------
     1024                                       70656                    1/2 5/2
                                                        (- 1/23 + 2/23 6   )

                        1/2  1/4                          1/2  1/4
                       24    6                13          4    6
     + 5/17664 ------------------------- + ----- -------------------------
                               1/2 5/2      256                    1/2 1/2
               (- 1/23 + 2/23 6   )               (- 1/23 + 2/23 6   )
```

```
          83    1/2  1/4                  1/2 1/2
      + ------ 4    6    (- 1/23 + 2/23 6   )   )
         1536

    1/2
   n

                           1/2 n   1/2
      1 / ((- 1/23 + 2/23 6   )  Pi   )

                  1
   + O(-----------------------)
                        1/2 n
         (- 1/23 + 2/23 6   )
```

Average cost for diff2 on random inputs of size n is:

av_tau_diff2_n :=

```
   16

                                                                  1/2  1/4
         115                   1/2 3/2  1/2  1/4     125           4    6
      (------ (- 1/23 + 2/23 6   )    4    6    - ------- -------------------------
        1024                                        70656                   1/2 5/2
                                                           (- 1/23 + 2/23 6   )

                              1/2  1/4                        1/2  1/4
                             24    6               13          4    6
        + 5/17664 ------------------------- + ----- -------------------------
                                    1/2 5/2     256                   1/2 1/2
                  (- 1/23 + 2/23 6   )              (- 1/23 + 2/23 6   )

             83    1/2  1/4                  1/2 1/2
        + ------ 4    6    (- 1/23 + 2/23 6   )   )
           1536

                         1/2 1/2  2
        (- 1/23 + 2/23 6   )    n

             1/2  1/4
        / (4    6   )

           3/2
      + O(n   )
```

Floating point evaluation :

```
                                          2      3/2
                                 .4117439326 n  + O(n   )
```

Function diff3:
Number of inputs of diff3 of size n is:

```
                                1/2 (- n)   1/2  1/4
                 (- 1/23 + 2/23 6   )      23    6                      1
             1/4 ------------------------------------ + O(--------------------------)
                                1/2 1/2  3/2   1/2                           1/2 n  2
                       (- 1 + 2 6   )    n    Pi              (- 1/23 + 2/23 6   )  n
```

```
Total cost for diff3 on random inputs of size n is:
     88107                 1                    3109               1
 (---------- ---------------------------- + ---------- ----------------------) n
   102981488                   1/2 4  3/2     205962976                  1/2 4
              (- 1/23 + 2/23 6   )  6                    (- 1/23 + 2/23 6   )
-------------------------------------------------------------------------------
                                      1/2 n
                                (- 1/23 + 2/23 6   )

   + 2

               1/2                 1/2 1/2                          1/2
              24    (- 1/23 + 2/23 6   )          2005              24
      (9/512 ------------------------------ - ------- ------------------------------
                          5/4                 11776                   1/2 1/2  5/4
                         6                             (- 1/23 + 2/23 6   )    6

                                    1/2                                    1/2
           24379                   24                    113               4
        + ---------- ------------------------------ + ------- ------------------------------
           12459008    5/4                  1/2 5/2     32768    7/4                  1/2 5/2
                      6    (- 1/23 + 2/23 6   )                6    (- 1/23 + 2/23 6   )

                            1/2                                    1/2
                           24                                     24
        + 1/8192 ------------------------------ + 1/23552 ------------------------------
                  7/4                  1/2 5/2              5/4                  1/2 7/2
                 6    (- 1/23 + 2/23 6   )                 6    (- 1/23 + 2/23 6   )

                            1/2                          1/2                  1/2 1/2
            2005            4                     225    4    (- 1/23 + 2/23 6   )
        + ------- ------------------------------ - ------ ------------------------------
           23552                   1/2 1/2  5/4     2048                5/4
                    (- 1/23 + 2/23 6   )    6                          6

                            1/2                          3/2                  1/2 3/2
                           4                     1977   4    (- 1/23 + 2/23 6   )
        - 1/47104 ------------------------------ + ------ ------------------------------
                   5/4                  1/2 7/2     2048                3/4
                  6    (- 1/23 + 2/23 6   )                           6

                            1/2                                    3/2
            1403            4                     161               4
        - ------- ------------------------------ + ----- ------------------------------
           32768    7/4                  1/2 1/2     512    3/4                  1/2 1/2
                   6    (- 1/23 + 2/23 6   )                6    (- 1/23 + 2/23 6   )

                                 1/2 5/2  1/2                          3/2
           38617   (- 1/23 + 2/23 6   )    4          11               4
        - ------- ------------------------------ + ------ ------------------------------
           16384                  7/4                 1024    3/4                  1/2 3/2
                                 6                           6    (- 1/23 + 2/23 6   )

                    1/2                 1/2 3/2                          1/2
           46115   4    (- 1/23 + 2/23 6   )          241373              4
        - ------- ------------------------------ - ---------- ------------------------------
           32768                  7/4                 12459008    5/4                  1/2 5/2
                                 6                              6    (- 1/23 + 2/23 6   )
```

```
                  1/2                  1/2 7/2                              1/2 1/2  3/2
     36501   4    (- 1/23 + 2/23 6   )         609   (- 1/23 + 2/23 6   )    4
- ------- ---------------------------- + ------ ----------------------------
     32768                7/4                 1024                3/4
                         6                                        6

                         3/2                                   1/2 1/2  1/2
      21                4                     1679   (- 1/23 + 2/23 6   )    4
- ------ ---------------------------- - ------ ----------------------------)
     2048    3/4                  1/2 5/2     8192                7/4
            6    (- 1/23 + 2/23 6   )                            6

   1/2
  n

                          1/2 n   1/2
    1 / ((- 1/23 + 2/23 6   )  Pi   )

             1
+ O(----------------------)
                    1/2 n
     (- 1/23 + 2/23 6   )
```

Average cost for diff3 on random inputs of size n is:

```
                                                1/2    1/2  5/2
                                       (29369 + 3109 6   ) Pi    n          2
              av_tau_diff3_n := 1/184 ---------------------------------- + O(n )
                                                 1/2 7/2  3/4   1/2
                                       (- 1 + 2 6   )    6    23
```

Floating point evaluation :

```
                                          5/2      2
                                .1655658473 n    + O(n )
```

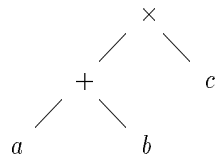
$\Lambda\Upsilon\Omega$ Report 7

Average case analysis of distributivity

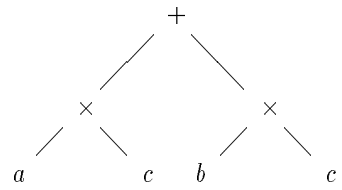
I Problem specification

Many simple symbolic manipulation algorithms fall in the category of strict top–down algorithms. Such is the case for symbolic differentiation. On the other hand "simplification" algorithms operate often in a bottom up or mixed fashion, as is illustrated by the case of the simplification rule $X \times 0 \rightarrow 0$: In that case, a "deep" 0 may cancel factors that are much higher in the tree.

This report presents a partial analysis of the distributivity rule, used classically in the `expand` primitive of computer algebra systems. We consider here the *left–distributivity* rule by which an expression



transforms into



We see that the symbol $\times$ goes down and that $+$ goes up in the tree. If we have a long branch of $\times$ with a $+$ at the end, this $+$ will "cross" all $\times$. Hence we must detect (arbitrary long) branches of $\times$ followed by a $+$. Such a programme can not be written using $\Lambda\Upsilon\Omega$. What we can do instead is write some "approximation programmes", i.e. programmes that look at a finite depth k.

Let us consider here a programme that detects *one* $\times$ followed by a $+$ (i.e. $k = 2$). The natural way of writing such a programme in Adl would be:

```
type expression = x
                | plus(expression,expression)
                | times(expression,expression);

function distrib(e:expression):expression;
case e of
  plus(t,u)          : plus(distrib(t),distrib(u));
  times(plus(v,w),u) : plus(distrib(times(v,u)),distrib(times(w,u)));
```

```
  times(t,u)          : times(distrib(t),distrib(u));
  x                   : x;
end;
```

Unfortunately, this programme can not be processed in the current stage of ALAS. The difficulty is that in the call `distrib(times(v,u))`, we call `distrib` on a *subtype* of `expression`. Hence the complexity descriptor of this call is not $\tau distrib(z)$. To bypass the problem, we must give a name to this subtype and write a special function for it. This was done in the Adl programme below.

The average cost of our approximation to full distributivity, `distrib`, is found to behave like:

$$Cn^{3/2}A^n + O(nA^n)$$

where $C \simeq 0.09944211381$ and $A \simeq 1/\sqrt{0.1238993431 \times 8} = 1.004431917$. This example is interesting since `distrib` has an exponential cost (with a small constant: quantity A^n exceeds $n^{3/2}$ only for $n \simeq 2700$!).

II Source program (Adl)

```
type expression = x
                | plus(expression,expression)
                | expression1;
     expression1 = times(expression,expression);
     plus,times,x = atom(1);

function distrib(e:expression):expression;
case e of
  plus(t,u)        : plus(distrib(t),distrib(u));
  times(t,u)       : distrib1(e);
  x                : x
end;

function distrib1(e:expression1):expression;
case e of
  times(t,u):
    case t of
      plus(v,w)   : plus(distrib1(times(v,u)),
                         distrib1(times(w,u)));
      times(v,w): times(distrib1(t),distrib(u));
      x           : times(x,distrib(u))
    end
end;

measure plus,times,x : 1;

to_analyze : distrib;
```

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
  expression(z)=x(z)+plus(z)*expression(z)*expression(z)+expression1(z)
  expression1(z)=times(z)*expression(z)*expression(z)
```

```
  plus(z)=z
  times(z)=z
  x(z)=z

Complexity descriptors:
  tau_distrib(z)=0*plus(z)*expression(z)*expression(z)+1*plus(z)*expression(z)*expression(z)+plus(z
)*tau_distrib(z)*expression(z)/1+plus(z)*expression(z)*tau_distrib(z)/1+tau_distrib1(z)/1+1*x(z)+0
  tau_distrib1(z)=0*times(z)*plus(z)*expression(z)*expression(z)*expression(z)+1*times(z)*plus(z)*e
xpression(z)*expression(z)*expression(z)+times(z)*plus(z)*tau_distrib1(z)*expression(z)*1/times(z)+
times(z)*plus(z)*expression(z)*tau_distrib1(z)*1/times(z)+0*times(z)*times(z)*expression(z)*express
ion(z)*expression(z)+1*times(z)*times(z)*expression(z)*expression(z)*expression(z)+times(z)*tau_dis
trib1(z)*expression(z)/1+times(z)*times(z)*expression(z)*expression(z)*tau_distrib(z)/1+0*times(z)*
x(z)*expression(z)+1*times(z)*x(z)*expression(z)+1*times(z)*x(z)*expression(z)+times(z)*x(z)*tau_di
strib(z)/1+0+0
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...

tau_distrib(z)
```

$$= -\left(-\frac{1}{16}\,\frac{\left(-1+\left(1-8z\right)^{2\,1/2}\right)^{2}}{z} - \frac{1}{4}\,z\left(-1+\left(1-8z\right)^{2\,1/2}\right) - z\right)$$

$$-\frac{1}{64}\,\frac{\left(-1+\left(1-8z\right)^{2\,1/2}\right)^{3}}{z}$$

$$1 \Big/ \left(-\frac{1}{4}+\frac{5}{4}\left(1-8z\right)^{2\,1/2} + \frac{5}{16}\left(-1+\left(1-8z\right)^{2\,1/2}\right)^{2} - z\right)$$

V Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Function distrib:
Number of inputs of distrib of size n is:
```

$$\frac{8^{(1/2\,n)}}{(1/2)^{3/2}\,n^{3/2}\,Pi^{1/2}} + O\left(4^{(1/2\,n)}\,\frac{1}{(1/8)^{2}\,n^{2}}\right)$$

```
for n mod 2 = 1, and 0 otherwise.

Total cost for distrib on random inputs of size n is:
                                                Let

s[   1   ]=   .1238993431
Root of    -3-5*(1-8*z)**(1/2)+28*z   =0
                                          then we get :
```

```
                                                                        1/2
                          1                                    (1 - 8 s[1])
(- 1/2 ------------------------------------- + 5/8 ------------------------------------
                    s[1]                                 s[1]
       (- 20 ----------------- - 28 s[1]) s[1]      (- 20 ----------------- - 28 s[1]) s[1]
                      1/2                                  1/2
             (1 - 8 s[1])                         (1 - 8 s[1])

                                                        1/2
                   1                                s[1]
  + ---------------------------------- - 6 ----------------------------------
                s[1]                               s[1]
     - 20 ----------------- - 28 s[1]     - 20 ----------------- - 28 s[1]
                    1/2                                1/2
          (1 - 8 s[1])                       (1 - 8 s[1])

               1/2            1/2                                3/2
          s[1]    (1 - 8 s[1])                      (1 - 8 s[1])
  - 2 ---------------------------------- - 1/8 ---------------------------------------)
                s[1]                               s[1]
     - 20 ----------------- - 28 s[1]      (- 20 ----------------- - 28 s[1]) s[1]
                    1/2                                1/2
           (1 - 8 s[1])                       (1 - 8 s[1])

         n
  1 / s[1]

         1
  + O(---------)
          n
     s[1]  n
```

Or

$$.1586866546\ (.1238993431)^{(-1/2)\,n} + O\left(2\ (.1238993431)^{(1/2)\,n}\right)^{-\frac{1}{n}}$$

for n mod 2 = 1, and 0 otherwise.
Average cost for distrib on random inputs of size n is:

Let

s[1]= .1238993431
Root of -3-5*(1-8*z)**(1/2)+28*z =0

then we get :

```
                                                                        1/2
                          1                                    (1 - 8 s[1])
(- 1/2 ------------------------------------- + 5/8 ------------------------------------
                    s[1]                                 s[1]
       (- 20 ----------------- - 28 s[1]) s[1]      (- 20 ----------------- - 28 s[1]) s[1]
                      1/2                                  1/2
             (1 - 8 s[1])                         (1 - 8 s[1])

                                                        1/2
                   1                                s[1]
  + ---------------------------------- - 6 ----------------------------------
                s[1]                               s[1]
     - 20 ----------------- - 28 s[1]     - 20 ----------------- - 28 s[1]
                    1/2                                1/2
          (1 - 8 s[1])                       (1 - 8 s[1])
```

```
                1/2             1/2                                  3/2
           s[1]    (1 - 8 s[1])                         (1 - 8 s[1])
   - 2 ---------------------------------- - 1/8 ----------------------------------------)
                  s[1]                                   s[1]
       - 20 ----------------- - 28 s[1]        (- 20 ----------------- - 28 s[1]) s[1]
                       1/2                                    1/2
             (1 - 8 s[1])                           (1 - 8 s[1])
```

```
     n   1/2  3/2
(1/8)  Pi    n
```

```
          n
   1 / s[1]
```

```
                                                 1/2                        3/2
                          1                    (-7)                       (-7)
   + O(- (- 11/2 ---------------------- - 11/8 ---------------------- - 1/8 ----------------------)
                          1                      1                          1
                - 20 --------- - 28          - 20 --------- - 28        - 20 --------- - 28
                         1/2                    1/2                        1/2
                      (-7)                   (-7)                       (-7)
```

```
          n   1/2
     (1/8)  Pi    n)
```

```
                                        Or
```

```
                                      (- 1/2 n)  (- 1/2 n)   1/2      3/2  3/2
av_tau_distrib_n := .1586866546 (.1238993431)          8          Pi    (1/2)    n
```

```
   + O(1/2
```

```
                                                 1/2                        3/2
                        1                      (-7)                       (-7)
      (- 11/2 ---------------------- - 11/8 ---------------------- - 1/8 ----------------------)
                        1                      1                          1
              - 20 --------- - 28          - 20 --------- - 28        - 20 --------- - 28
                       1/2                    1/2                        1/2
                    (-7)                   (-7)                       (-7)
```

```
             (1/2 n)
        (1/8)        n)
```

```
for n mod 2 = 1, and 0 otherwise.
Floating point evaluation :
```

```
                         (- .5000000000 n)   (- .5000000000 n)  3/2
.09944211381 (.1238993431)                 8.                  n
```

```
   + O(1/2
```

```
                                                 1/2                        3/2
                        1                      (-7)                       (-7)
      (- 11/2 ---------------------- - 11/8 ---------------------- - 1/8 ----------------------)
                        1                      1                          1
              - 20 --------- - 28          - 20 --------- - 28        - 20 --------- - 28
                       1/2                    1/2                        1/2
                    (-7)                   (-7)                       (-7)
```

```
             (1/2 n)
        (1/8)        n)
```

ΛγΩ Report 8

Average case analysis of mutually recursive functions

I Problem specification

This example shows, in a simple context, the analysis of term rewriting systems along the lines of [CKS87]. Different systems are treated in the next chapter. Here, we consider a rewriting system with mutually recursive operators introduced in [CKS87]. The algebra of terms (binary trees) is obtained from one binary constructor `o` and one constant `a`:

$$\mathcal{T} = a \quad \cup \quad o(\mathcal{T}, \mathcal{T})$$

The rewriting operators, f_1 and f_2, are defined by the following rules:

$$\begin{aligned} f_1(a) &= a \\ f_1(o(x,y)) &= o(o(a, f_2(x,y)), a) \end{aligned}$$

$$\begin{aligned} f_2(a,a) &= o(a,a) \\ f_2(o(x,y),a) &= o(o(f_1(x), f_1(y)), a) \\ f_2(a, o(x,y)) &= o(a, f_2(x,y)) \\ f_2(o(x_1,y_1), o(x_2,y_2)) &= o(o(f_1(x_1), f_1(y_2)), o(f_1(y_1), f_1(x_2))) \end{aligned}$$

The complexity of this system over a given expression is the number of steps required to obtain the normal form. In complexity terms, this is the number of calls to f_1 and f_2.

The analysis result is: f_1 and f_2 have an asymptotic average cost that is linear. More precisely, the average cost of f_1 (resp. f_2) over a term of size n (resp. over two terms of *total* size n) behaves like:

$$\frac{5}{7}n + O(\sqrt{n}) \quad n \to \infty.$$

II Source program (Adl)

```
type tree = a | o(tree,tree);
     a,o = atom(1);

function f1(t:tree):tree;
begin
  c1;
  case t of
    a            : a;
```

```
    o(x,y)          : o(o(a,f2(x,y)),a);
  end;
end;

function f2(t,u:tree):tree;
begin
  c2;
  case t,u of
    a,a                 : o(a,a);
    (o,x,y),a           : o(o(f1(x),f1(y)),a);
    a,(o,x,y)           : o(a,f2(y,x));
    (o,x1,y1),o(x2,y2)  : o(o(f1(x1),f1(y2)),o(f1(y1),f1(x2)));
  end;
end;

measure c1,c2:1;
```

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
  tree(z)=a(z)+o(z)*tree(z)*tree(z)
  a(z)=z
  o(z)=z

Complexity descriptors:
  tau_f1(z)=1*tree(z)+0*a(z)+0*o(z)*tree(z)*tree(z)+0*o(z)*tree(z)*tree(z)+0*o(z)*tree(z)*tree(z)+0
*o(z)*tree(z)*tree(z)+0*o(z)*tree(z)*tree(z)+o(z)*tau_f2(z)*1+0*o(z)*tree(z)*tree(z)+0+0
  tau_f2(z)=1*tree(z)*tree(z)+0*a(z)*a(z)+0*a(z)*a(z)+0*a(z)*a(z)+0*a(z)*a(z)+0*o(z)*tree(z)*tree(z
)*a(z)+0*o(z)*tree(z)*tree(z)*a(z)+0*o(z)*tree(z)*tree(z)*a(z)+0*o(z)*tree(z)*tree(z)*a(z)+o(z)*tau
_f1(z)*tree(z)*a(z)/1+o(z)*tree(z)*tau_f1(z)*a(z)/1+0*o(z)*tree(z)*tree(z)*a(z)+0*a(z)*o(z)*tree(z)
*tree(z)+0*a(z)*o(z)*tree(z)*tree(z)+0*a(z)*o(z)*tree(z)*tree(z)+a(z)*o(z)*1*tau_f2(z)+0*o(z)*tree(
z)*tree(z)*o(z)*tree(z)*tree(z)+0*o(z)*tree(z)*tree(z)*o(z)*tree(z)*tree(z)+0*o(z)*tree(z)*tree(z)*
o(z)*tree(z)*tree(z)+0*o(z)*tree(z)*tree(z)*o(z)*tree(z)*tree(z)+o(z)*tau_f1(z)*tree(z)*o(z)*tree(z
)*tree(z)/1+o(z)*tree(z)*tree(z)*o(z)*tree(z)*tau_f1(z)/1+0*o(z)*tree(z)*tree(z)*o(z)*tree(z)*tree(
z)+0*o(z)*tree(z)*tree(z)*o(z)*tree(z)*tree(z)+o(z)*tree(z)*tau_f1(z)*o(z)*tree(z)*tree(z)/1+o(z)*t
ree(z)*tree(z)*o(z)*tau_f1(z)*tree(z)/1+0+0
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...
```

$$\mathrm{tau\_f1}(z) = -\frac{1/2\,\left(-1+\left(1-4z\right)^{1/2}\right)\left(-3/2+z+1/2\left(1-4z\right)^{1/2}\right)}{z\left(-1-1/2\left(-1+\left(1-4z\right)^{1/2}\right)+z-z\left(-1+\left(1-4z\right)^{1/2}\right)\right)}$$

```
                                        2 1/2 2                 2 1/2 2          2
                              (- 1 + (1 - 4 z )   ) ((- 1 + (1 - 4 z )   ) + 1 + 2 z )
tau_f2(z) = - 1/4 ------------------------------------------------------------------------
                    2                            2 1/2 3    2    2               2 1/2
                   z  (- 1 - 1/2 (- 1 + (1 - 4 z )   ) + z  - z  (- 1 + (1 - 4 z )   ))
```

V Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Function f1:
Number of inputs of f1 of size n is:
                                  (1/2 n)
                                 4                          1
                          ---------------------- + O(4 -----------------)
                                3/2  3/2   1/2              (1/2 n)  2
                           (1/2)    n    Pi            (1/4)        n

for n mod 2 = 1, and 0 otherwise.

Total cost for f1 on random inputs of size n is:
                                    (1/2 n)
                                   4                            1
                        10/7 ---------------------- + O(2 -----------------)
                                  1/2  1/2   1/2              (1/2 n)
                             (1/2)    n    Pi            (1/4)        n

for n mod 2 = 1, and 0 otherwise.
Average cost for f1 on random inputs of size n is:

                                                          1/2  1/2
                              av_tau_f1_n := 5/7 n + O((1/2)    n   )

for n mod 2 = 1, and 0 otherwise.
Floating point evaluation :

                                                       1/2  1/2
                                 .7142857143 n + O((1/2)    n   )

Function f2:
Number of inputs of f2 of size n is:
                                  (1/2 n)
                                 4                          1
                          ---------------------- + O(4 -----------------)
                                3/2  3/2   1/2              (1/2 n)  2
                           (1/2)    n    Pi            (1/4)        n

for n mod 2 = 0, and 0 otherwise.

Total cost for f2 on random inputs of size n is:
                                    (1/2 n)
                                   4                            1
                        10/7 ---------------------- + O(2 -----------------)
                                  1/2  1/2   1/2              (1/2 n)
                             (1/2)    n    Pi            (1/4)        n

for n mod 2 = 0, and 0 otherwise.
Average cost for f2 on random inputs of size n is:
```

$$av\_tau\_f2\_n := 5/7\ n + O((1/2)^{1/2}\ n^{1/2})$$

for n mod 2 = 0, and 0 otherwise.
Floating point evaluation :

$$.7142857143\ n + O((1/2)^{1/2}\ n^{1/2})$$

ΛΥΩ Report 9

Average case analysis of Shuffles of Trees

I Problem specification

These examples are taken from [CKS87] and they belong to the field of rewriting systems. Three cases are considered, leading to average $O(1)$, $O(n)$ and $O(C^n)$ complexity.

Expressions are represented by binary trees, and there are three sets of rules, each defining a particular shuffle on these trees. Denote by $o(a,b)$ a binary tree whose left and right (root) subtrees are a and b, and use capital letters for leaves (terminal symbols in expressions). The three cases treated are defined below.

I.1 First set of rules

$$\begin{aligned}
shuffle(A,B) &= o(A,B)\\
shuffle(A,b) &= o(A,b)\\
shuffle(a,B) &= o(a,B)\\
shuffle(o(a,b),o(c,d)) &= o(shuffle(a,c),shuffle(b,d))
\end{aligned}$$

I.2 Second set of rules

$$\begin{aligned}
shuffle(A,B) &= o(A,B)\\
shuffle(A,b) &= o(A,b)\\
shuffle(a,B) &= o(a,B)\\
shuffle(o(a,b),o(c,d)) &= o(o(shuffle(a,c),shuffle(a,d)),\\
&\quad\ o(shuffle(b,c),shuffle(b,d)))
\end{aligned}$$

I.3 Third set of rules

$$\begin{aligned}
shuffle(A,B) &= o(A,B)\\
shuffle(A,b) &= o(A,b)\\
shuffle(a,B) &= o(a,B)\\
shuffle(o(a,b),o(c,d)) &= o(o(o(shuffle(a,c),shuffle(c,a)),\\
&\quad\ o(shuffle(a,d),shuffle(d,a))),\\
&\quad\ o(o(shuffle(b,c),shuffle(c,b)),\\
&\quad\ o(shuffle(b,d),shuffle(d,b))))
\end{aligned}$$

This is implemented in $\Lambda\Upsilon^{\Omega}$ in a straightforward way: Specify a type 'bintree' for binary trees, and a function 'shuffle' which takes two arguments and constructs inductively the shuffle of two trees.

The results show that small modifications on the definitions of the functions strongly affect their behaviours. In a way, the complexity phenomena that this example illustrates are the analog of extinction, criticality and explosion in branching processes.

II Source program (Adl)

```
type bintree = leaf | o(bintree,bintree);
     leaf,o = atom(1);

function shuffle1(x,y:bintree):bintree;
case x,y of
  A,B                : begin count; o(A,B) end;
  A,o(a,b)           : begin count; o(A,o(a,b)) end;
  o(a,b),B           : begin count; o(o(a,b),B) end;
  o(a,b),o(c,d)      : begin count; o(shuffle1(a,c),shuffle1(b,d)) end;
end;

function shuffle2(x,y:bintree):bintree;
case x,y of
  A,B                : begin count; o(A,B) end;
  A,o(a,b)           : begin count; o(A,o(a,b)) end;
  o(a,b),B           : begin count; o(o(a,b),B) end;
  o(a,b),o(c,d)      : begin count; o(o(shuffle2(a,c),shuffle2(a,d)),
                                      o(shuffle2(b,c),shuffle2(b,d))) end;
end;

function shuffle3(x,y:bintree):bintree;
case x,y of
  A,B                : begin count; o(A,B) end;
  A,o(a,b)           : begin count; o(A,o(a,b)) end;
  o(a,b),B           : begin count; o(o(a,b),B) end;
  o(a,b),o(c,d)      : begin count; o(o(o(shuffle3(a,c),shuffle3(c,a)),
                                       o(shuffle3(a,d),shuffle3(d,a))),
                                     o(o(shuffle3(b,c),shuffle3(c,b)),
                                       o(shuffle3(b,d),shuffle3(d,b)))) end;

end;

measure count : 1;
```

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
  bintree(z)=leaf(z)+o(z)*bintree(z)*bintree(z)
  leaf(z)=z
  o(z)=z

Complexity descriptors:
```

```
  tau_shuffle1(z)=1*bintree(z)*bintree(z)+0*bintree(z)*bintree(z)+0*bintree(z)*bintree(z)+0*bintree
(z)*bintree(z)+0*bintree(z)*bintree(z)+1*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bi
ntree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)
+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*
bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+1*o(z)*bintree(z)*bintree(z)*bintree(
z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z
)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintre
e(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+1*o(z)*bintree
(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0
*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+o(z)*tau_shuffle1(z)*bintree(z)*o(z)*1*bintr
ee(z)+o(z)*bintree(z)*tau_shuffle1(z)*o(z)*bintree(z)*1+0
  tau_shuffle2(z)=1*bintree(z)*bintree(z)+0*bintree(z)*bintree(z)+0*bintree(z)*bintree(z)+0*bintree
(z)*bintree(z)+0*bintree(z)*bintree(z)+1*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bi
ntree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)
+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*
bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+1*o(z)*bintree(z)*bintree(z)*bintree(
z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z
)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintre
e(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+1*o(z)*bintree
(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0
*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)
*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+o(z)*tau_shuffle2(z)*bintree(z)
*o(z)*1*bintree(z)+o(z)*tau_shuffle2(z)*bintree(z)*o(z)*bintree(z)*1+0*o(z)*bintree(z)*bintree(z)*o
(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+o(z)*bintree(z)*t
au_shuffle2(z)*o(z)*1*bintree(z)+o(z)*bintree(z)*tau_shuffle2(z)*o(z)*bintree(z)*1+0
  tau_shuffle3(z)=1*bintree(z)*bintree(z)+0*bintree(z)*bintree(z)+0*bintree(z)*bintree(z)+0*bintree
(z)*bintree(z)+0*bintree(z)*bintree(z)+1*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bi
ntree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)
+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*
bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+1*o(z)*bintree(z)*bintree(z)*bintree(
z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z
)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintre
e(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+1*o(z)*bintree
(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0
*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)
*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+o(z)*tau_shuffle3(z)*bintree(z)
*o(z)*1*bintree(z)+o(z)*1*bintree(z)*o(z)*tau_shuffle3(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)
*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+o(z)*
*tau_shuffle3(z)*bintree(z)*o(z)*bintree(z)*1+o(z)*1*bintree(z)*o(z)*bintree(z)*tau_shuffle3(z)+0*o
(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*b
intree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)
*bintree(z)*bintree(z)+o(z)*bintree(z)*tau_shuffle3(z)*o(z)*1*bintree(z)+o(z)*bintree(z)*1*o(z)*tau
_shuffle3(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*b
intree(z)*o(z)*bintree(z)*bintree(z)+o(z)*bintree(z)*tau_shuffle3(z)*o(z)*bintree(z)*1+o(z)*bintree
(z)*1*o(z)*bintree(z)*tau_shuffle3(z)+0
```

IV Solving equations

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
  bintree(z)=leaf(z)+o(z)*bintree(z)*bintree(z)
  leaf(z)=z
  o(z)=z
```

```
Complexity descriptors:
  tau_shuffle1(z)=1*bintree(z)*bintree(z)+0*bintree(z)*bintree(z)+0*bintree(z)*bintree(z)+0*bintree
(z)*bintree(z)+0*bintree(z)*bintree(z)+1*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bi
ntree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)
+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*
bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+1*o(z)*bintree(z)*bintree(z)*bintree(
z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z
)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintre
e(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+1*o(z)*bintree
(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0
*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+o(z)*tau_shuffle1(z)*bintree(z)*o(z)*1*bintr
ee(z)+o(z)*bintree(z)*tau_shuffle1(z)*o(z)*bintree(z)*1+0
  tau_shuffle2(z)=1*bintree(z)*bintree(z)+0*bintree(z)*bintree(z)+0*bintree(z)*bintree(z)+0*bintree
(z)*bintree(z)+0*bintree(z)*bintree(z)+1*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bi
ntree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)
+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*
bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+1*o(z)*bintree(z)*bintree(z)*bintree(
z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z
)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintre
e(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+1*o(z)*bintree
(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0
*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)
*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+o(z)*tau_shuffle2(z)*bintree(z)
*o(z)*1*bintree(z)+o(z)*tau_shuffle2(z)*bintree(z)*o(z)*bintree(z)*1+0*o(z)*bintree(z)*bintree(z)*o
(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+o(z)*bintree(z)*t
au_shuffle2(z)*o(z)*1*bintree(z)+o(z)*bintree(z)*tau_shuffle2(z)*o(z)*bintree(z)*1+0
  tau_shuffle3(z)=1*bintree(z)*bintree(z)+0*bintree(z)*bintree(z)+0*bintree(z)*bintree(z)+0*bintree
(z)*bintree(z)+0*bintree(z)*bintree(z)+1*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bi
ntree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)
+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+0*bintree(z)*o(z)*
bintree(z)*bintree(z)+0*bintree(z)*o(z)*bintree(z)*bintree(z)+1*o(z)*bintree(z)*bintree(z)*bintree(
z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z
)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintre
e(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*bintree(z)+1*o(z)*bintree
(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0
*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)
*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(
z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+o(z)*tau_shuffle3(
z)*bintree(z)*o(z)*1*bintree(z)+o(z)*1*bintree(z)*o(z)*tau_shuffle3(z)*bintree(z)+0*o(z)*bintree(z)
*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+o(z)
*tau_shuffle3(z)*bintree(z)*o(z)*bintree(z)*1+o(z)*1*bintree(z)*o(z)*bintree(z)*tau_shuffle3(z)+0*o
(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*b
intree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)
*bintree(z)*bintree(z)+o(z)*bintree(z)*tau_shuffle3(z)*o(z)*1*bintree(z)+o(z)*bintree(z)*1*o(z)*tau
_shuffle3(z)*bintree(z)+0*o(z)*bintree(z)*bintree(z)*o(z)*bintree(z)*bintree(z)+0*o(z)*bintree(z)*b
intree(z)*o(z)*bintree(z)*bintree(z)+o(z)*bintree(z)*tau_shuffle3(z)*o(z)*bintree(z)*1+o(z)*bintree
(z)*1*o(z)*bintree(z)*tau_shuffle3(z)+0
```

V Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...

tau_shuffle1(z)
```

```
                              2 1/2 2                          2 1/2 3
                 (- 1 + (1 - 4 z )   )            (- 1 + (1 - 4 z )   )
    = - (- 1/4 ------------------------ + 1/4 ------------------------
                           2                                2
                          z                                z

                             2 1/2 4
                    (- 1 + (1 - 4 z )   )
      - 1/16 ------------------------)
                           2
                          z

                              2 1/2 2
      / (1 - 1/2 (- 1 + (1 - 4 z )   ) )
```

tau_shuffle2(z)

```
                              2 1/2 2                          2 1/2 3
                 (- 1 + (1 - 4 z )   )            (- 1 + (1 - 4 z )   )
    = - (- 1/4 ------------------------ + 1/4 ------------------------
                           2                                2
                          z                                z

                             2 1/2 4
                    (- 1 + (1 - 4 z )   )
      - 1/16 ------------------------)
                           2
                          z

                             2 1/2 2
      / (1 - (- 1 + (1 - 4 z )   ) )
```

tau_shuffle3(z)

```
                              2 1/2 2                          2 1/2 3
                 (- 1 + (1 - 4 z )   )            (- 1 + (1 - 4 z )   )
    = - (- 1/4 ------------------------ + 1/4 ------------------------
                           2                                2
                          z                                z

                             2 1/2 4
                    (- 1 + (1 - 4 z )   )
      - 1/16 ------------------------)
                           2
                          z

                              2 1/2 2
      / (1 - 2 (- 1 + (1 - 4 z )   ) )
```

VI Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...
```

```
Function shuffle1:
Number of inputs of shuffle1 of size n is:
                                  (1/2 n)
                                 4                          1
                          ---------------------- + O(4 -----------------)
                                3/2  3/2   1/2              (1/2 n)  2
                           (1/2)    n    Pi            (1/4)        n
```

```
for n mod 2 = 0, and 0 otherwise.

Total cost for shuffle1 on random inputs of size n is:
                                  (1/2 n)
                                 4                              1
                      21/2 ---------------------- + O(4 ------------------)
                                 3/2  3/2   1/2              (1/2 n)  2
                           (1/2)    n    Pi             (1/4)        n

for n mod 2 = 0, and 0 otherwise.
Average cost for shuffle1 on random inputs of size n is:

                                                              1
                          av_tau_shuffle1_n := 21/2 + O(---------------)
                                                            1/2  1/2
                                                       (1/2)    n

for n mod 2 = 0, and 0 otherwise.
Floating point evaluation :

                                                    1
                                  10.50000000 + O(---------------)
                                                          1/2  1/2
                                                     (1/2)    n

Function shuffle2:
Number of inputs of shuffle2 of size n is:
                                   (1/2 n)
                                  4                              1
                          ---------------------- + O(4 ------------------)
                                3/2  3/2   1/2              (1/2 n)  2
                           (1/2)    n    Pi             (1/4)        n

for n mod 2 = 0, and 0 otherwise.

Total cost for shuffle2 on random inputs of size n is:
                                (1/2 n)
                               4                                1
                     9/8 ---------------------- + O(-----------------------------)
                               1/2  1/2   1/2             (1/2 n)      3/2  3/2
                          (1/2)    n    Pi           (1/4)        (1/2)    n

for n mod 2 = 0, and 0 otherwise.
Average cost for shuffle2 on random inputs of size n is:

                                                            1/2  1/2
                          av_tau_shuffle2_n := 9/16 n + O((1/2)    n   )

for n mod 2 = 0, and 0 otherwise.
Floating point evaluation :

                                                      1/2  1/2
                                  .5625000000 n + O((1/2)    n   )

Function shuffle3:
Number of inputs of shuffle3 of size n is:
                                   (1/2 n)
                                  4                              1
                          ---------------------- + O(4 ------------------)
                                3/2  3/2   1/2              (1/2 n)  2
                           (1/2)    n    Pi             (1/4)        n
```

```
for n mod 2 = 0, and 0 otherwise.

Total cost for shuffle3 on random inputs of size n is:
  185                                      1
(----- ------------------------------------------------------------------------------
   64                                                        1/2
                      1/2             1                     2                    1/2
        (- 1/8 + 1/4 2   ) (- ----------------- + 2 ----------------- + 1 - 2 2   )
                                      1/2 1/2               1/2 1/2
                             (3/2 - 2   )          (3/2 - 2   )

                                           1/2 1/2
                                    (3/2 - 2   )
   - 9/4 ------------------------------------------------------------------------------
                                                          1/2
                      1/2             1                     2                    1/2
        (- 1/8 + 1/4 2   ) (- ----------------- + 2 ----------------- + 1 - 2 2   )
                                       1/2 1/2               1/2 1/2
                              (3/2 - 2   )          (3/2 - 2   )

                                          1/2
      25                                  2
   - ---- ---------------------------------------------------------------------------
      16                                                  1/2
                       1/2             1                     2                    1/2
         (- 1/8 + 1/4 2   ) (- ----------------- + 2 ----------------- + 1 - 2 2   )
                                        1/2 1/2               1/2 1/2
                               (3/2 - 2   )          (3/2 - 2   )

                                        1/2 1/2  1/2
                                 (3/2 - 2   )    2
   + 1/2 ------------------------------------------------------------------------------)
                                                          1/2
                      1/2             1                     2                    1/2
        (- 1/8 + 1/4 2   ) (- ----------------- + 2 ----------------- + 1 - 2 2   )
                                       1/2 1/2               1/2 1/2
                              (3/2 - 2   )          (3/2 - 2   )

                  1/2 (1/2 n)
   1 / (- 1/8 + 1/4 2   )

                     1
   + O(2 ----------------------------)
                     1/2 (1/2 n)
          (- 1/8 + 1/4 2   )        n

for n mod 2 = 0, and 0 otherwise.
Average cost for shuffle3 on random inputs of size n is:

av_tau_shuffle3_n :=

     185                                      1
   (----- ------------------------------------------------------------------------------
      64                                                  1/2
                       1/2             1                     2                    1/2
         (- 1/8 + 1/4 2   ) (- ----------------- + 2 ----------------- + 1 - 2 2   )
                                        1/2 1/2               1/2 1/2
                               (3/2 - 2   )          (3/2 - 2   )
```

```
                                                  1/2 1/2
                                          (3/2 - 2   )
      - 9/4 ------------------------------------------------------------------------------
                                                                  1/2
                                                                 2                    1/2
                          1/2            1                                          1/2
            (- 1/8 + 1/4 2   ) (- ----------------- + 2 ----------------- + 1 - 2 2   )
                                        1/2 1/2              1/2 1/2
                                 (3/2 - 2   )          (3/2 - 2   )

                                                 1/2
         25                                     2
      - ---- ------------------------------------------------------------------------------
         16                                                       1/2
                                                                 2                    1/2
                          1/2            1                                          1/2
            (- 1/8 + 1/4 2   ) (- ----------------- + 2 ----------------- + 1 - 2 2   )
                                        1/2 1/2              1/2 1/2
                                 (3/2 - 2   )          (3/2 - 2   )

                                                    1/2 1/2  1/2
                                           (3/2 - 2   )    2
      + 1/2 ------------------------------------------------------------------------------)
                                                                  1/2
                                                                 2                    1/2
                          1/2            1                                          1/2
            (- 1/8 + 1/4 2   ) (- ----------------- + 2 ----------------- + 1 - 2 2   )
                                        1/2 1/2              1/2 1/2
                                 (3/2 - 2   )          (3/2 - 2   )

          (1/2 n)   1/2      3/2  3/2
     (1/4)        Pi    (1/2)    n

                          1/2 (1/2 n)
      1 / (- 1/8 + 1/4 2   )

                           (1/2 n)
                      (1/4)        n
      + O(1/2 ---------------------------)
                               1/2 (1/2 n)
               (- 1/8 + 1/4 2   )
```

for n mod 2 = 0, and 0 otherwise.
Floating point evaluation :

```
                                    (.5000000000 n)  3/2                        (1/2 n)
                        (.2500000000)                n                      (1/4)        n
        .1422491652 ------------------------------------ + O(1/2 ---------------------------)
                                     (.5000000000 n)                              1/2 (1/2 n)
                          (.2285533905)                             (- 1/8 + 1/4 2   )
```

$\Lambda\Upsilon\Omega$ Report 10

Average case analysis of a function over regular expressions

I Problem specification

We examine here a problem that has served as a classical example in the study of automatic program transformation, namely determining the collection of letters that can occur as initial letters in a regular language generated by a regular expression.

Regular expressions are: either the empty string denoted by λ, or a letter, or a repetition —Kleene star operation— of a regular expression $(A^\star)$, or the union $(A + B)$, or the concatenation $(A \cdot B)$ of two regular expressions. In this example, due to F. Vivares [Viv88], we analyze a function, *head*(E), which determines all possible first letters of words in a language defined by a regular expression E. The inductive definition is

$$\begin{aligned} \mathrm{head}(\lambda) &= \emptyset \\ \mathrm{head}(x) &= \{x\} \\ \mathrm{head}(A^\star) &= \mathrm{head}(A) \\ \mathrm{head}(A+B) &= \mathrm{head}(A) \cup \mathrm{head}(B) \\ \mathrm{head}(A \cdot B) &= \mathrm{head}(A) \cup \delta(A) \cdot \mathrm{head}(B), \end{aligned}$$

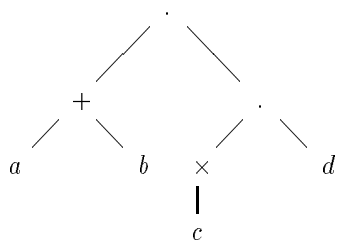
where *delta* (δ) determines if an expression may produce the empty string (1 stands for *true* and 0 for *false*):

$$\begin{aligned} \delta(x) &= 0 \\ \delta(\lambda) &= 1 \\ \delta(A^\star) &= 1 \\ \delta(A+B) &= \delta(A) + \delta(B) \\ \delta(A \cdot B) &= \delta(A) \cdot \delta(B). \end{aligned}$$

Given a random regular expression with n symbols, how many times will both functions be called? The worst case is in $n^2/4$, for expressions defined by:

$$A_1 = \begin{array}{ccc} & \cdot & \\ / & & \backslash \\ \lambda & & \lambda \end{array} \qquad A_{k+1} = \begin{array}{ccc} & \cdot & \\ / & & \backslash \\ A_k & & \lambda \end{array}$$

which satisfy $size(A_k) = 2k+1$ and $\tau\mathrm{head}(A_k) = k^2+4k-1$. If we take the above system as a rewriting system, we consider the number of (rewriting) steps required to obtain the normal form of a regular expression. For *head* (resp. *delta*), this normal form is a set of letters (resp. 0 or 1). For example, the expression:



needs 7 calls (4 to *head* and 3 to *delta*), assuming that in the rule for $head(A \cdot B)$, we do not compute $head(B)$ when $\delta(A) = 0$. In the strict version of $\Lambda\Upsilon^{\Omega}$, we are not allowed to use instructions like

$$\textbf{if}\ \delta(A)\ \textbf{then}\ \mathrm{head}(B).$$

But if we design a function *head_lower* that does the same as *head*, except for $(A \cdot B)$ where it calls only $head\_lower(A)$ and $delta(A)$, the complexity of *head_lower* will be a lower bound on the complexity of *head*. Similarly, we can also design a function *head_upper*. In the Adl programme below, we have chosen a model with only one letter, denoted by `letter`.

When we operate over random regular expressions as arguments, the average cost of *head_lower* (resp. *head_upper*) given by $\Lambda\Upsilon^{\Omega}$ is $145/4 + O(1/\sqrt{n})$ (resp. $3n/2 + O(\sqrt{n})$). Therefore we conclude that the average number of calls to *head* and *delta* in order to compute the first letters of a regular expression of size n satisfies:

$$\frac{145}{4} + O(\frac{1}{\sqrt{n}}) \leq \tau\mathrm{head}_n \leq \frac{3n}{2} + O(\sqrt{n}).$$

The exact average cost of $\tau\mathrm{head}_n$ has been computed independently by hand, thanks to the fact that the counting generating function for expressions for which *delta* yields 1 is computable. This cost turns out to be asymptotically constant:

$$\tau\mathrm{head}_n = \frac{1185}{4} + O(\frac{1}{\sqrt{n}}).$$

II Source program (Adl)

```
type regular_expression = lambda | letter
                        | seq(regular_expression)
                        | plus(regular_expression,regular_expression)
                        | prod(regular_expression,regular_expression);
     lambda,letter,seq,plus,prod = atom(1);

procedure delta(r : regular_expression);
case r of
    letter      : count;
    lambda      : count;
    seq(A)      : count;
    plus(A,B)   : begin count; delta(A); delta(B) end;
    prod(A,B)   : begin count; delta(A) end;
end;

procedure head_lower(r : regular_expression);       % lower bound %
case r of
    letter      : count;
    lambda      : count;
    seq(A)      : begin count; head_lower(A) end;
    plus(A,B)   : begin count; head_lower(A); head_lower(B) end;
    prod(A,B)   : begin count; head_lower(A); delta(A) end;
end;
```

```
procedure head_upper(r : regular_expression);        % upper bound %
case r of
    letter        : count;
    lambda        : count;
    seq(A)        : begin count; head_upper(A) end;
    plus(A,B)     : begin count; head_upper(A); head_upper(B) end;
    prod(A,B)     : begin count; head_upper(A); delta(A); head_upper(B) end;
end;

measure count : 1;

to_analyze : delta, head_lower, head_upper;
```

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
  regular_expression(z)=lambda(z)+letter(z)+seq(z)*regular_expression(z)+plus(z)*regular_expression
(z)*regular_expression(z)+prod(z)*regular_expression(z)*regular_expression(z)
  lambda(z)=z
  letter(z)=z
  seq(z)=z
  plus(z)=z
  prod(z)=z

Complexity descriptors:
  tau_delta(z)=1*letter(z)+1*lambda(z)+1*seq(z)*regular_expression(z)+1*plus(z)*regular_expression(
z)*regular_expression(z)+plus(z)*tau_delta(z)*regular_expression(z)/1+plus(z)*regular_expression(z)
*tau_delta(z)/1+1*prod(z)*regular_expression(z)*regular_expression(z)+prod(z)*tau_delta(z)*regular_
expression(z)/1+0
  tau_head_lower(z)=1*letter(z)+1*lambda(z)+1*seq(z)*regular_expression(z)+seq(z)*tau_head_lower(z)
/1+1*plus(z)*regular_expression(z)*regular_expression(z)+plus(z)*tau_head_lower(z)*regular_expressi
on(z)/1+plus(z)*regular_expression(z)*tau_head_lower(z)/1+1*prod(z)*regular_expression(z)*regular_e
xpression(z)+prod(z)*tau_head_lower(z)*regular_expression(z)/1+prod(z)*tau_delta(z)*regular_express
ion(z)/1+0
  tau_head_upper(z)=1*letter(z)+1*lambda(z)+1*seq(z)*regular_expression(z)+seq(z)*tau_head_upper(z)
/1+1*plus(z)*regular_expression(z)*regular_expression(z)+plus(z)*tau_head_upper(z)*regular_expressi
on(z)/1+plus(z)*regular_expression(z)*tau_head_upper(z)/1+1*prod(z)*regular_expression(z)*regular_e
xpression(z)+prod(z)*tau_head_upper(z)*regular_expression(z)/1+prod(z)*tau_delta(z)*regular_express
ion(z)/1+prod(z)*regular_expression(z)*tau_head_upper(z)/1+0
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...

tau_delta(z)

                                                                    2 1/2 2
                                        2 1/2        (- 1 + z + (1 - 2 z - 15 z )   )
      - 7/4 z - 1/4 + 1/4 (1 - 2 z - 15 z )    - 1/8 ---------------------------------
```

$$= - \frac{z}{\left(1/4 + 3/4\, z + 3/4\, \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}\right)^{2 \, 1/2}}$$

tau_head_lower(z)

$$= - \left(- 7/4\, z - 1/4 + 1/4\, \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2} - 1/8\, \frac{\left(- 1 + z + \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}\right)^{2 \, 1/2 \, 2}}{z}\right.$$

$$+ 1/2\, \frac{z\, \left(- 1 + z + \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}\right)}{\left(1/4 + 3/4\, z + 3/4\, \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}\right)^{2 \, 1/2}}$$

$$- 1/16\, \frac{\left(- 1 + z + \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}\right)^{2 \, 1/2 \, 2}}{\left(1/4 + 3/4\, z + 3/4\, \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}\right)^{2 \, 1/2}}$$

$$\left. + 1/32\, \frac{\left(- 1 + z + \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}\right)^{2 \, 1/2 \, 3}}{z\, \left(1/4 + 3/4\, z + 3/4\, \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}\right)^{2 \, 1/2}}\right)$$

$$/ \left(1/4 - 1/4\, z + 3/4\, \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}\right)$$

tau_head_upper(z)

$$= - \left(- 7/4\, z - 1/4 + 1/4\, \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2} - 1/8\, \frac{\left(- 1 + z + \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}\right)^{2 \, 1/2 \, 2}}{z}\right.$$

$$+ 1/2\, \frac{z\, \left(- 1 + z + \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}\right)}{\left(1/4 + 3/4\, z + 3/4\, \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}\right)^{2 \, 1/2}}$$

$$- 1/16\, \frac{\left(- 1 + z + \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}\right)^{2 \, 1/2 \, 2}}{\left(1/4 + 3/4\, z + 3/4\, \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}\right)^{2 \, 1/2}}$$

$$\left. + 1/32\, \frac{\left(- 1 + z + \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}\right)^{2 \, 1/2 \, 3}}{z\, \left(1/4 + 3/4\, z + 3/4\, \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}\right)^{2 \, 1/2}}\right)$$

$$/ \left(1 - 2\, z - 15\, z\right)^{2 \, 1/2}$$

V Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Function delta:
Number of inputs of delta of size n is:
                                      1/2  n
                                    10    5             1
                                1/4 ------------ + O(-----------)
                                     3/2   1/2           n  2
                                    n    Pi          (1/5)  n

Total cost for delta on random inputs of size n is:
                                    (n - 1/2)  1/2
                                125   5          2              1
                               ----- ----------------- + O(-----------)
                                 16       3/2   1/2             n  2
                                         n    Pi            (1/5)  n

Average cost for delta on random inputs of size n is:

                                                              1
                                  av_tau_delta_n := 25/4 + O(------)
                                                             1/2
                                                            n

Floating point evaluation :

                                                    1
                                         6.250000000 + O(------)
                                                       1/2
                                                      n

Function head_lower:
Number of inputs of head_lower of size n is:
                                      1/2  n
                                    10    5             1
                                1/4 ------------ + O(-----------)
                                     3/2   1/2           n  2
                                    n    Pi          (1/5)  n

Total cost for head_lower on random inputs of size n is:
                                    (n - 1/2)  1/2
                                725   5          2              1
                               ----- ----------------- + O(-----------)
                                 16       3/2   1/2             n  2
                                         n    Pi            (1/5)  n

Average cost for head_lower on random inputs of size n is:

                                                                1
                               av_tau_head_lower_n := 145/4 + O(------)
                                                               1/2
                                                              n

Floating point evaluation :

                                                    1
                                         36.25000000 + O(------)
                                                       1/2
                                                      n
```

```
Function head_upper:
Number of inputs of head_upper of size n is:
                                        1/2  n
                                      10    5             1
                                1/4 ------------ + O(-----------)
                                      3/2   1/2            n  2
                                     n    Pi          (1/5)  n

Total cost for head_upper on random inputs of size n is:
                                      (1/2 + n)
                                     5                       1
                              3/4 ----------------- + O(-------------)
                                    1/2  1/2   1/2            n  3/2
                                   2    n    Pi          (1/5)  n

Average cost for head_upper on random inputs of size n is:

                                                              1/2
                               av_tau_head_upper_n := 3/2 n + O(n   )

Floating point evaluation :

                                                    1/2
                                  1.500000000 n + O(n   )
```

Part C

Combinatorial Problems

$\Lambda\Upsilon\Omega$ Report 11

Average case analysis of 2-Regular Graphs

I Problem specification

An undirected graph is said to be *2–regular* if each node has degree 2. We consider here the problem of estimating the expected number of connected components in a random *labelled* 2–regular graph of size n. The computation below demonstrates that this number is

$$\frac{1}{2}\log n + O(1). \tag{1}$$

This $\Lambda\Upsilon\Omega$ example shows how to estimate parameters of combinatorial structures by expressing them as complexity measures of suitable procedures.

The counting of k–regular graphs is in general a difficult problem [GJ83]. Graphs that are 2–regular are simpler: Clearly, such a graph is a collection of (undirected) cycles, each of which has length at least 3. They also have the interest of being in bijective correspondence with "clouds" [Com74]. A cloud is a configuration of intersection points of n lines in general position satisfying: (i) It has maximal (i.e. equal to n) cardinality; (ii) no 3 points are colinear.

The exact counting of clouds, or equivalently 2–regular graphs is done by Comtet in his book [Com74]. From our specification of 2–regular graphs, we find directly (Section 4) that the exponential generating function (EGF) of 2-regular graphs is

$$\exp\left(\frac{1}{2}\log\frac{1}{1-z} - \frac{z}{2} - \frac{z^2}{4}\right) \equiv \frac{e^{-z/2-z^2/4}}{\sqrt{1-z}}. \tag{2}$$

Notice that Comtet uses instead a recurrence principle followed by generating function calculations to attain this goal.

The asymptotic counting is done by Comtet as an illustration of Darboux's method. It is performed here automatically (Section 5) using "singularity analysis". We find that the number of 2–regular graphs of size n satisfies the asymptotic estimate

$$n! \cdot \frac{e^{-3/4}}{\sqrt{\pi n}}\left(1 + O(\frac{1}{n})\right). \tag{3}$$

The analysis in this report also includes the determination of the expected number of components. This is expressed by a programme that traverses a random 2–regular graphs and triggers a dummy procedure `count` each time it encounters a component. The corresponding expected value, namely $\frac{1}{2}\log n$, was obtained by [FS89] (Proposition 1 and Example 3) where it is also shown that the number of components has a limiting Gaussian distribution.

II Source program (Adl)

```
type
        TwoRegG = set(Component);
        Component = ucycle(Node,card>=3);
        Node=Latom(1);

procedure Visit(g:TwoRegG);
        forall c in g do
                count;
% end Visit %

measure count : 1;
```

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are exponential.

Counting generating functions:
  TwoRegG(z)=exp(Component(z))
  Component(z)=(L(Node(z))-(Node(z)+Node(z)^2/2))/2
  Node(z)=z

Complexity descriptors:
  tau_Visit(z)=(exp(Component(z))*1*Component(z))
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...
```

$$\text{tau\_Visit}(z) = \frac{\exp(-1/2\,z - 1/4\,z^2)\,(-1/2\,\ln(1-z) - 1/2\,z - 1/4\,z^2)}{(1-z)^{1/2}}$$

V Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Function Visit:
Number of arguments of Visit of size n is n! times:
```

$$\frac{\exp(-3/4)}{n^{1/2}\,Pi^{1/2}} + O\left(\frac{1}{n^{3/2}}\right)$$

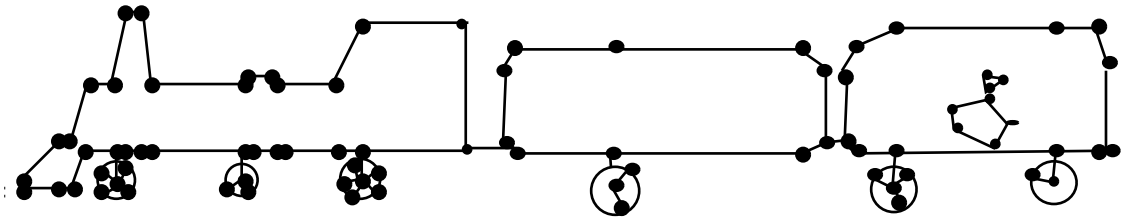
```
Total cost of Visit on all arguments of size n is n! times:
                                    exp(-3/4) ln(n)          1
                                1/2 ---------------- + O(------)
                                          1/2  1/2          1/2
                                        Pi    n            n

Average cost for Visit on random inputs of size n is:

                                av_tau_Visit_n := 1/2 ln(n) + O(1)

Floating point evaluation :

                                     .5000000000 ln(n) + O(1)
```

$\Lambda\Upsilon\Omega$ Report 12

Average case analysis of Random Trains

I Problem specification

This example is taken from [Fla85] where it was introduced in order to illustrate the power and generality of the "admissible constructor" method. Here we analyze this example automatically, and show how to use $\Lambda\Upsilon\Omega$ in order to determine the asymptotic behaviour of a parameter of a structure, by writing an algorithm whose cost is equal to this parameter. Here is the picture of a random train:



In the drawing above, imagine the dots as being labelled by distinct integers from 1 to the "size" of the train (its number of points). A train is described by the following structural equations:

$$\begin{cases} train = loco,\ wagons \\ wagons = sequence(wagon) \\ loco = sequence(slice) \\ slice = (upper, lower) \text{ or } (upper, lower, wheel) \\ wagon = loco,\ passengers \\ passengers = set(passenger) \\ passenger = (head, belly) \\ wheel = cycle(wheel\_element) \\ head = belly = cycle(passenger\_element) \\ upper = lower = wheel\_element = passenger\_element = point. \end{cases}$$

From this description, two trains bearing the same labels, and whose passengers are in the same wagon but in different positions will be counted only once. On the opposite, wheels are defined as cycles of point, hence two wheels in matching positions in two trains that are not equivalent modulo a circular permutation will induce different train structures.

In this session we shall consider the following questions:

How many trains of size n are there as n tends to infinity ?

What is the average position of the first wagon without passengers when n tends to infinity ?

To answer the second question, we have to build an algorithm whose cost is the number of occupied wagons it is necessary to walk through before entering an empty wagon. The general way to do that in $\Lambda\Upsilon\Omega$

is to write procedures which traverse the structure until the relevant level is attained, marking the parameter of interest by a call to a void function, say *count*, to which we associate the measure 1.

From the analytic point of view, the difficulty in this example is to handle properly a singularity which cannot be found explicitly. As the following analysis shows, $\Lambda\Upsilon^{\Omega}$ obtains the result that the first empty wagon is asymptotically at a constant distance from the locomotive ($c_1 = 0.70367$). A closely related scheme could be used to compute the position of the first occupied wagon, leading to another constant ($c_2 = 1.42112$). A numerical check shows with a good precision that $c_1 \cdot c_2 \approx 1$, a particular case of a general property of random sequences (corresponding distributions are asymptotically geometric).

II Source program (Adl)

```
type
    train = product(locomotive,wagons);
    wagons = sequence(wagon);
    locomotive = sequence(slice,card >= 1);
    slice = product(upper,lower)
          | product(upper,lower,wheel);
    wheel = product(center,cycle(wheel_element));
    wagon = product(locomotive,passengers);
    passengers = set(passenger);
    passenger = product(head,belly);
    head,belly = cycle(passenger_element);
    upper,lower,center,wheel_element,passenger_element = Latom(1);

function first_empty_wagon (t:train):integer;
case t of
  (loco,ws) : few(ws)
end;

procedure few(ws:wagons);
case ws of
  () : nil;
  ((l,sp),rest): F(sp,rest);
   otherwise: nil;
end;


procedure F(sp:passengers; rest:wagons);
case sp of
  {}              : nil;
  otherwise       : begin count; few(rest) end;
end;


measure count:1;
        nil:0;

to_analyze : first_empty_wagon;
```

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...
```

```
Generating functions are exponential.

Counting generating functions:
  train(z)=locomotive(z)*wagons(z)
  wagons(z)=Q(wagon(z))
  locomotive(z)=slice(z)^1*Q(slice(z))
  slice(z)=upper(z)*lower(z)+upper(z)*lower(z)*wheel(z)
  wheel(z)=center(z)*L(wheel_element(z))
  wagon(z)=locomotive(z)*passengers(z)
  passengers(z)=exp(passenger(z))
  passenger(z)=head(z)*belly(z)
  head(z)=L(passenger_element(z))
  belly(z)=L(passenger_element(z))
  upper(z)=z
  lower(z)=z
  center(z)=z
  wheel_element(z)=z
  passenger_element(z)=z

Complexity descriptors:
  tau_first_empty_wagon(z)=locomotive(z)*tau_few(z)/1+0
  tau_few(z)=0*1+locomotive(z)*tau_F(z)*1+0
  tau_F(z)=0*1*wagons(z)+1*(exp(passenger(z))-1)*wagons(z)+(exp(passenger(z))-1)*tau_few(z)/1
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...

tau_first_empty_wagon(z)

                   2                          2  4
    = (exp(ln(1 - z) ) - 1) (- 1 + z ln(1 - z))  z

              2              2     3                        2
      / (1 - z  exp(ln(1 - z) ) + z  ln(1 - z) exp(ln(1 - z) ))

              2    3              2              2     3                        2
      / (1 - z  + z  ln(1 - z) - z  exp(ln(1 - z) ) + z  ln(1 - z) exp(ln(1 - z) ))
```

V Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Function first_empty_wagon:
Number of arguments of first_empty_wagon of size n is n! times:
                                                Let

s[   1   ]=   .5180547070
Root of    1-z**2+z**3*ln(1-z)-z**2*exp(ln(1-z)**2)+z**3*ln(1-z)*exp(ln(1-z)**2)   =0
                                           then we get :

     2
(s[1]
```

```
                                                4
          2         3                     s[1]            2                  2
  / (2 s[1]  - 3 s[1]  ln(1 - s[1]) + ---------- + 2 s[1]  exp(ln(1 - s[1]) )
                                      1 - s[1]

             3                 2
          s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])          3                               2
    - 2 ---------------------------------------- - 3 s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) )
                       1 - s[1]

           4                 2            4             2                  2
        s[1]  exp(ln(1 - s[1]) )       s[1]  ln(1 - s[1])  exp(ln(1 - s[1]) )
    + ------------------------- + 2 ---------------------------------------)
                1 - s[1]                                1 - s[1]

       3
  - s[1]  ln(1 - s[1])

                                                  4
            2         3                     s[1]            2                  2
     / (2 s[1]  - 3 s[1]  ln(1 - s[1]) + ---------- + 2 s[1]  exp(ln(1 - s[1]) )
                                        1 - s[1]

               3                 2
            s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])          3                               2
       - 2 ---------------------------------------- - 3 s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) )
                         1 - s[1]

              4                 2            4             2                  2
           s[1]  exp(ln(1 - s[1]) )       s[1]  ln(1 - s[1])  exp(ln(1 - s[1]) )
       + ------------------------- + 2 ---------------------------------------))
                   1 - s[1]                                1 - s[1]

         n
  1 / s[1]

          1
  + O(---------)
           n
       s[1]  n
```

Or

$$.1008557594\ (.5180547070)^{(-n)} + O\left(\frac{1}{(.5180547070)^{n}\ n}\right)$$

Total cost of first_empty_wagon on all arguments of size n is n! times:

Let

s[1]= .5180547070

Root of 1-z**2+z**3*ln(1-z)-z**2*exp(ln(1-z)**2)+z**3*ln(1-z)*exp(ln(1-z)**2) =0

then we get :

```
                2      4
(exp(ln(1 - s[1]) ) s[1]

             2                 2        3                               2
  / (1 - s[1]  exp(ln(1 - s[1]) ) + s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) ))
```

```
                                                4
        2         3                          s[1]            2                  2
/ (2 s[1]  - 3 s[1]  ln(1 - s[1]) + ---------- + 2 s[1]  exp(ln(1 - s[1]) )
                                     1 - s[1]

           3                 2
        s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])          3                               2
   - 2 ---------------------------------------- - 3 s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) )
                      1 - s[1]

         4                 2            4             2                  2
      s[1]  exp(ln(1 - s[1]) )       s[1]  ln(1 - s[1])  exp(ln(1 - s[1]) )
   + ------------------------ + 2 ---------------------------------------)
             1 - s[1]                                1 - s[1]

-2

                  2      5
   exp(ln(1 - s[1]) ) s[1]  ln(1 - s[1])

               2                 2        3                                2
      / (1 - s[1]  exp(ln(1 - s[1]) ) + s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) ))

                                                4
             2         3                    s[1]            2                  2
      / (2 s[1]  - 3 s[1]  ln(1 - s[1]) + ---------- + 2 s[1]  exp(ln(1 - s[1]) )
                                          1 - s[1]

                 3                 2
              s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])          3                               2
         - 2 ---------------------------------------- - 3 s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) )
                            1 - s[1]

               4                 2            4             2                  2
            s[1]  exp(ln(1 - s[1]) )       s[1]  ln(1 - s[1])  exp(ln(1 - s[1]) )
         + ------------------------ + 2 ---------------------------------------)
                    1 - s[1]                                1 - s[1]

                  2      6             2
+ exp(ln(1 - s[1]) ) s[1]  ln(1 - s[1])

              2                 2        3                                2
   / (1 - s[1]  exp(ln(1 - s[1]) ) + s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) ))

                                                4
           2         3                    s[1]            2                  2
   / (2 s[1]  - 3 s[1]  ln(1 - s[1]) + ---------- + 2 s[1]  exp(ln(1 - s[1]) )
                                       1 - s[1]

              3                 2
           s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])          3                               2
      - 2 ---------------------------------------- - 3 s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) )
                         1 - s[1]

            4                 2            4             2                  2
         s[1]  exp(ln(1 - s[1]) )       s[1]  ln(1 - s[1])  exp(ln(1 - s[1]) )
      + ------------------------ + 2 ---------------------------------------)
                1 - s[1]                                1 - s[1]

      4
- s[1]
```

```
            2                  2        3                                2
   / (1 - s[1]  exp(ln(1 - s[1]) ) + s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) ))

                                                4
          2         3                          s[1]            2                  2
   / (2 s[1]  - 3 s[1]  ln(1 - s[1]) + ---------- + 2 s[1]  exp(ln(1 - s[1]) )
                                       1 - s[1]

             3                 2
          s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])          3                               2
      - 2 ---------------------------------------- - 3 s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) )
                        1 - s[1]

            4                  2             4             2                  2
         s[1]  exp(ln(1 - s[1]) )        s[1]  ln(1 - s[1])  exp(ln(1 - s[1]) )
      + ------------------------- + 2 ---------------------------------------)
                1 - s[1]                                1 - s[1]

+ 2

       5
   s[1]  ln(1 - s[1])

                2                  2        3                                2
      / (1 - s[1]  exp(ln(1 - s[1]) ) + s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) ))

                                                  4
            2         3                          s[1]            2                  2
      / (2 s[1]  - 3 s[1]  ln(1 - s[1]) + ---------- + 2 s[1]  exp(ln(1 - s[1]) )
                                          1 - s[1]

               3                 2
            s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])          3                               2
        - 2 ---------------------------------------- - 3 s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) )
                          1 - s[1]

              4                  2             4             2                  2
           s[1]  exp(ln(1 - s[1]) )        s[1]  ln(1 - s[1])  exp(ln(1 - s[1]) )
        + ------------------------- + 2 ---------------------------------------)
                  1 - s[1]                                1 - s[1]

      6             2
- s[1]  ln(1 - s[1])

            2                  2        3                                2
   / (1 - s[1]  exp(ln(1 - s[1]) ) + s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) ))

                                                4
          2         3                          s[1]            2                  2
   / (2 s[1]  - 3 s[1]  ln(1 - s[1]) + ---------- + 2 s[1]  exp(ln(1 - s[1]) )
                                       1 - s[1]

             3                 2
          s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])          3                               2
      - 2 ---------------------------------------- - 3 s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) )
                        1 - s[1]

            4                  2             4             2                  2
         s[1]  exp(ln(1 - s[1]) )        s[1]  ln(1 - s[1])  exp(ln(1 - s[1]) )
      + ------------------------- + 2 ---------------------------------------))
                1 - s[1]                                1 - s[1]
```

```
        n
1 / s[1]

        1
+ O(---------)
         n
     s[1]  n
```

Or

```
                                        (- n)              1
                  .07097007911 (.5180547070)      + O(------------------)
                                                            n
                                                 (.5180547070)  n
```

Average cost for first_empty_wagon on random inputs of size n is:

Let

```
s[   1   ]=   .5180547070
Root of    1-z**2+z**3*ln(1-z)-z**2*exp(ln(1-z)**2)+z**3*ln(1-z)*exp(ln(1-z)**2)   =0
                                      then we get :
```

```
                2      4
(exp(ln(1 - s[1]) ) s[1]

             2                 2        3                              2
   / (1 - s[1]  exp(ln(1 - s[1]) ) + s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) ))

                                          4
          2         3                  s[1]            2                2
   / (2 s[1]  - 3 s[1]  ln(1 - s[1]) + ---------- + 2 s[1]  exp(ln(1 - s[1]) )
                                      1 - s[1]

             3                2
          s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])          3                              2
     - 2 ---------------------------------------- - 3 s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) )
                       1 - s[1]

           4                 2            4             2                 2
        s[1]  exp(ln(1 - s[1]) )       s[1]  ln(1 - s[1])  exp(ln(1 - s[1]) )
     + ------------------------- + 2 ----------------------------------------)
               1 - s[1]                              1 - s[1]

   -2

                    2      5
      exp(ln(1 - s[1]) ) s[1]  ln(1 - s[1])

                 2                 2        3                              2
         / (1 - s[1]  exp(ln(1 - s[1]) ) + s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) ))

                                             4
                2         3                s[1]            2                2
         / (2 s[1]  - 3 s[1]  ln(1 - s[1]) + ---------- + 2 s[1]  exp(ln(1 - s[1]) )
                                           1 - s[1]

                    3                2
                 s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])          3                              2
            - 2 ---------------------------------------- - 3 s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) )
                              1 - s[1]
```

```
                4                 2            4             2                 2
             s[1]  exp(ln(1 - s[1]) )       s[1]  ln(1 - s[1])  exp(ln(1 - s[1]) )
          + ------------------------- + 2 ------------------------------------------)
                      1 - s[1]                          1 - s[1]

                   2      6             2
+ exp(ln(1 - s[1]) ) s[1]  ln(1 - s[1])

              2                 2        3                              2
   / (1 - s[1]  exp(ln(1 - s[1]) ) + s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) ))

                                              4
            2         3                    s[1]            2                 2
   / (2 s[1]  - 3 s[1]  ln(1 - s[1]) + ---------- + 2 s[1]  exp(ln(1 - s[1]) )
                                       1 - s[1]

                3                 2
             s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])          3                              2
      - 2 ---------------------------------------- - 3 s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) )
                          1 - s[1]

              4                 2            4             2                 2
             s[1]  exp(ln(1 - s[1]) )       s[1]  ln(1 - s[1])  exp(ln(1 - s[1]) )
      + ------------------------- + 2 ------------------------------------------)
                      1 - s[1]                          1 - s[1]

      4
- s[1]

              2                 2        3                              2
   / (1 - s[1]  exp(ln(1 - s[1]) ) + s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) ))

                                              4
            2         3                    s[1]            2                 2
   / (2 s[1]  - 3 s[1]  ln(1 - s[1]) + ---------- + 2 s[1]  exp(ln(1 - s[1]) )
                                       1 - s[1]

                3                 2
             s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])          3                              2
      - 2 ---------------------------------------- - 3 s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) )
                          1 - s[1]

              4                 2            4             2                 2
             s[1]  exp(ln(1 - s[1]) )       s[1]  ln(1 - s[1])  exp(ln(1 - s[1]) )
      + ------------------------- + 2 ------------------------------------------)
                      1 - s[1]                          1 - s[1]

+ 2

       5
   s[1]  ln(1 - s[1])

                2                 2        3                              2
      / (1 - s[1]  exp(ln(1 - s[1]) ) + s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) ))

                                                4
              2         3                    s[1]            2                 2
      / (2 s[1]  - 3 s[1]  ln(1 - s[1]) + ---------- + 2 s[1]  exp(ln(1 - s[1]) )
                                         1 - s[1]
```

```
                3                2
           s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])          3                              2
       - 2 ------------------------------------ - 3 s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) )
                          1 - s[1]

             4                2           4            2                2
          s[1]  exp(ln(1 - s[1]) )       s[1]  ln(1 - s[1])  exp(ln(1 - s[1]) )
       + ------------------------ + 2 ---------------------------------------)
                 1 - s[1]                          1 - s[1]

    6            2
- s[1]  ln(1 - s[1])

            2                2       3                            2
  / (1 - s[1]  exp(ln(1 - s[1]) ) + s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) ))

                                          4
          2        3                  s[1]           2                2
  / (2 s[1]  - 3 s[1]  ln(1 - s[1]) + ---------- + 2 s[1]  exp(ln(1 - s[1]) )
                                      1 - s[1]

             3                2
          s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])          3                              2
     - 2 ------------------------------------ - 3 s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) )
                        1 - s[1]

           4                2           4            2                2
        s[1]  exp(ln(1 - s[1]) )       s[1]  ln(1 - s[1])  exp(ln(1 - s[1]) )
     + ------------------------ + 2 ---------------------------------------))
               1 - s[1]                          1 - s[1]

     2
/ (s[1]

                                          4
          2        3                  s[1]           2                2
  / (2 s[1]  - 3 s[1]  ln(1 - s[1]) + ---------- + 2 s[1]  exp(ln(1 - s[1]) )
                                      1 - s[1]

             3                2
          s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])          3                              2
     - 2 ------------------------------------ - 3 s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) )
                        1 - s[1]

           4                2           4            2                2
        s[1]  exp(ln(1 - s[1]) )       s[1]  ln(1 - s[1])  exp(ln(1 - s[1]) )
     + ------------------------ + 2 ---------------------------------------)
               1 - s[1]                          1 - s[1]

       3
  - s[1]  ln(1 - s[1])

                                            4
            2        3                  s[1]           2                2
     / (2 s[1]  - 3 s[1]  ln(1 - s[1]) + ---------- + 2 s[1]  exp(ln(1 - s[1]) )
                                        1 - s[1]

               3                2
            s[1]  exp(ln(1 - s[1]) ) ln(1 - s[1])          3                              2
       - 2 ------------------------------------ - 3 s[1]  ln(1 - s[1]) exp(ln(1 - s[1]) )
                          1 - s[1]
```

```
                4                 2           4             2                 2
           s[1]  exp(ln(1 - s[1]) )       s[1]  ln(1 - s[1])  exp(ln(1 - s[1]) )
      + ------------------------- + 2 ------------------------------------------))
                 1 - s[1]                          1 - s[1]

 + O(1/n)
```

Or

```
av_tau_first_empty_wagon_n := .7036789920 + O(1/n)
```

Floating point evaluation :

```
.7036789920 + O(1/n)
```

$\Lambda\Upsilon\Omega$ Report 13

Average case analysis of Pollard's rho–method

I Problem specification

There are n^n *mappings* (functions) from a set E of cardinality n into itself. A number of applications related to the study of random number generators, integer factorisation and cryptography require determining expected values of parameters of random mappings. A mapping can be alternatively described as a *functional graph*. Nodes are elements of E; the set of edges that represents function f is the collection of ordered pairs $(x, f(x))$ for $x \in E$.

A functional graph is a collection of connected components. Each component is itself a cycle of planted trees. Many parameters relating to the iteration structure of f have simple interpretations on the associated functional graph.

Pollard's *rho–method* for integer factorisation is described in [Knu81]. In order to factor integer m, it iterates special polynomial functions

$$f(x) = x^2 + a \pmod{m}$$

where E is $\mathbf{Z}/m\mathbf{Z}$ and a is an arbitrary integer. The raw version of the algorithm is as follows

```
function pollard-rho(m : integer) : integer;
begin
     x0:=random(1..m);
     slow := x0; fast := x0;
     repeat
          slow := f(slow);
          fast := f(f(fast));
          d := GCD(fast - slow,m);
     until (d is_not_trivial);
     return(d);
end;
```

In order to analyze the complexity of the algorithm, we look at the iteration structure of f modulo a prime factor n of m. Since n is prime, function f has the property that —except for $y = a$— the set $f^{(-1)}(y)$ is of cardinality either 0 or 2. The main parameters are the distance λ of the starting value x_0 in E to its cycle, and length μ of the cycle seen from x_0. The complexity of the algorithm is linearly dependent on $\lambda + \mu$.

The algorithm motivates the study of random *binary functional graphs* (a node has in–degree either 0 or 2), each taken with equal likelihood. This is a heuristic model for analyzing Pollard's algorithm (it assumes that a polynomial behaves like a random function) and simulations confirm its empirical validity. We show

here that in such a random graph, a point is at expected distance

$$\sqrt{\frac{n}{8\pi}} + O(1) \tag{13.1}$$

from its cycle. Thus the expectation of λ is $O(\sqrt{n})$ and a similar estimate is established for the expectation of μ. Under the heuristic model, Pollard's rho method therefore operates in expected time $O(\sqrt{n})$, where n is the smallest prime factor of m.

In the type definitions below, we define binary functional graphs (`graphb`) as sets of components that are cycles of non–plane *labelled* binary trees. The parameter that we actually analyze is a generalised path–length (`graphpl`) representing the sum of distances of all points to their respective cycle. This is constructed inductively: First, we need a function which gives the size of a tree (`treesize`); next path–length in trees (`treepl`) is constructed in the usual way using its inductive definition; finally we cumulate path length over cycles (`componentpl`) and then components (`graphpl`).

The expected distance of a random point to a cycle is thus $1/n$ times the graph path–length (`graphpl`) which yields the expectation of the λ parameter (13.1). In passing, we showed that the number of binary functional graphs (`graphb`) of even size n is asymptotic to

$$2^{n/2}\sqrt{\frac{2}{\pi n}},$$

corresponding to a very simple generating function,

$$\frac{1}{\sqrt{1-2z^2}}.$$

Results from this analysis where first obtained "by hand" by Flajolet ([Fla81]).

II Source program (Adl)

```
type
    graphb = set(componentb);
    componentb = cycle(plantedtree);
    plantedtree = product(node,treeb);
    treeb = node | product(node,set(treeb,card=2));
    node = Latom(1);

procedure treesize(t:treeb);
begin
case t of
    node : count;
    (node,s) : begin count; forall x in s do treesize(x); end
end
end;

procedure pathlength(t:treeb);
begin
    treesize(t);
    case t of
        node : nil;
        (node,s) : forall x in s do pathlength(x)
    end
end;
```

```
procedure plantedtreepl(p:plantedtree);
begin
        case p of
            (node,x) : begin treesize(x); pathlength(x) end
        end
end;

procedure componentpl(c:componentb);
begin
        forall x in c do
            plantedtreepl(x);
end;

procedure graphpl(g:graphb);
begin
        forall c in g do
            componentpl(c);
end;


measure count:1;

to_analyze :  graphpl;
```

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are exponential.

Counting generating functions:
  graphb(z)=exp(componentb(z))
  componentb(z)=L(plantedtree(z))
  plantedtree(z)=node(z)*treeb(z)
  treeb(z)=node(z)+node(z)*treeb(z)^2/2!
  node(z)=z

Complexity descriptors:
  tau_treesize(z)=1*node(z)+1*node(z)*treeb(z)^2/2!+node(z)*(tau_treesize(z)/1*treeb(z)^1/1!)+0+0
  tau_pathlength(z)=tau_treesize(z)/1+0*node(z)+node(z)*(tau_pathlength(z)/1*treeb(z)^1/1!)+0
  tau_plantedtreepl(z)=node(z)*tau_treesize(z)/1+node(z)*tau_pathlength(z)/1+0
  tau_componentpl(z)=tau_plantedtreepl(z)/1/(1-plantedtree(z))+0
  tau_graphpl(z)=(exp(componentb(z))*tau_componentpl(z)/1)+0
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...
```

$$\text{tau\_graphpl}(z) = 2\,\frac{z^2}{(-1+2z)^{2\,2}}$$

V Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Function graphpl:
Number of arguments of graphpl of size n is n! times:
                                (1/2 n)
                               2                                1
                      ---------------------- + O(-----------------------------)
                            1/2  1/2   1/2            (1/2 n)      3/2  3/2
                       (1/2)    n    Pi          (1/2)        (1/2)    n

for n mod 2 = 0, and 0 otherwise.
Total cost of graphpl on all arguments of size n is n! times:
                                     (1/2 n)              1
                                1/2 2        n + O(--------------)
                                                      (1/2 n)
                                                   (1/2)

for n mod 2 = 0, and 0 otherwise.
Average cost for graphpl on random inputs of size n is:

                                          1/2      3/2  3/2          1/2  1/2
                    av_tau_graphpl_n := Pi    (1/2)    n    + O((1/2)    n   )

for n mod 2 = 0, and 0 otherwise.
Floating point evaluation :

                                        3/2          1/2  1/2
                                .6266570687 n    + O((1/2)    n   )
```

$\Lambda\Upsilon\Omega$ Report 14

Average case analysis of the pathlength variance

I Problem specification

This example shows that $\Lambda\Upsilon\Omega$ can also be of help in computing the variance of a cost distribution, and more generally, moments of any order k. We will compute here the variance of internal pathlength in binary trees. First define binary trees:

$$\mathcal{B} = \epsilon \cup o(\mathcal{B}, \mathcal{B}),$$

where $size(\epsilon) = 0$ and $size(o) = 1$. The internal pathlength of a binary tree is the sum of the distances of all internal nodes to the root. Hence it is be defined recursively by the following rules,

$$\begin{aligned} \text{pathlength}(\epsilon) &= 0 \\ \text{pathlength}(o(u,v)) &= \text{size}(u) + \text{size}(v) + \text{pathlength}(u) + \text{pathlength}(v), \end{aligned} \tag{14.1}$$

and size itself is defined recursively by

$$\begin{aligned} \text{size}(\epsilon) &= 0 \\ \text{size}(o(u,v)) &= 1 + \text{size}(u) + \text{size}(v). \end{aligned}$$

The pathlength variance over trees of size n is

$$V_n(\text{pathlength}) = E_n(\text{pathlength}^2) - E_n^2(\text{pathlength})$$

The second term of the right hand side is the square of the average pathlength. We already know how to compute it using $\Lambda\Upsilon\Omega$ (see Report 13 for a similar problem). The first term is the average of the square of the pathlength, which will be computed using the following complexity descriptor:

$$\tau\text{pathlength2} = \sum_{t \in \mathcal{T}} \text{pathlength}^2(t) z^{|t|}$$

It suffices now to square the right hand side of equation (14.1) and we obtain a 10 term sum for τpathlength2. Each of these terms represent a complexity descriptor, which will also be expressed recursively. In fact, using obvious symmetries, the problem reduces to considering only 6 different complexity descriptors:

$$\begin{aligned} \tau\text{pathlength2} &= \sum_{t \in \mathcal{T}} \text{pathlength}^2(t) z^{|t|} \\ \tau\text{size2} &= \sum_{t \in \mathcal{T}} \text{size}^2(t) z^{|t|} \end{aligned}$$

$$
\begin{aligned}
\tau\text{size\_x\_size} &= \sum_{u,v\in\mathcal{T}} \text{size}(u)\text{size}(v)z^{|u|+|v|} \\
\tau\text{size\_x\_pathlength} &= \sum_{u,v\in\mathcal{T}} \text{size}(u)\text{pathlength}(v)z^{|u|+|v|} \\
\tau\text{pathlength\_x\_pathlength} &= \sum_{u,v\in\mathcal{T}} \text{pathlength}(u)\text{pathlength}(v)z^{|u|+|v|} \\
\tau\text{size\_x\_pathlength1} &= \sum_{t\in\mathcal{T}} \text{size}(t)\text{pathlength}(t)z^{|t|}.
\end{aligned}
$$

The results given by $\Lambda\Upsilon^{\Omega}$ are:

$$
\begin{aligned}
\tau\text{pathlength}_n &= \sqrt{\pi}n^{3/2} + O(n) \\
\tau\text{pathlength2}_n &= \frac{10}{3}n^3 + O(n^{5/2})
\end{aligned}
$$

from which we deduce:

$$
\begin{aligned}
V\text{pathlength}_n &= \tau\text{pathlength2}_n - (\tau\text{pathlength}_n)^2 \\
&= (\frac{10}{3} - \pi)n^3 + O(n^{5/2}) = Cn^3 + O(n^{5/2})
\end{aligned}
$$

with $C \simeq 0.191740679$.

II Source program (Adl)

```
type bintree = epsilon | product(o, bintree, bintree);
     o = atom (1);

% size ( o(u,v) ) = 1 + size(u) + size(v) %

procedure size (t : bintree);
begin
  case t of
    ()          : nil;
    o(u,v)      : begin count; size(u); size(v); end;
  end;
end;


% pathlength ( o(u,v) ) = size(u) + size(v)
                        + pathlength(u) + pathlength(v) %

procedure pathlength (t : bintree);
begin
  case t of
    ()          : nil;
    o(u,v)      : begin size(u); size(v);
                        pathlength(u); pathlength(v);
                  end;
  end;
end;

% pathlength2 computes the square of the pathlength %
```

```
% size2 computes the square of the size                    %

% pathlength2 ( o(u,v) ) = size2(u) + size2(v) + pathlength2(u)
   + pathlength2(v) + 2 size(u) size(v) + 2 size(u) pathlength(u)
   + 2 size(u) pathlength(v) + 2 size(v) pathlength(u)
   + 2 size(v) pathlength(v) + 2 pathlength(u) pathlength(v) %

procedure pathlength2 (t : bintree);
begin
  case t of
    ()          : nil;
    o(u,v)      : begin size2(u); size2(v); pathlength2(u); pathlength2(v);
                        to 2 do begin
                          size_x_size(u,v); size_x_pathlength1(u);
                          size_x_pathlength(u,v); size_x_pathlength(v,u);
                          size_x_pathlength1(v); pathlength_x_pathlength(u,v);
                        end;
                  end;
  end;
end;


% size2 ( o(u,v) ) = 1 + size2(u) + size2(v) + 2 size(u) + 2 size(v)
                     + 2 size(u) size(v) %

procedure size2 (t : bintree);
begin
  case t of
    ()          : nil;
    o(u,v)      : begin count; size2(u); size2(v);
                        to 2 do begin size(u); size(v); size_x_size(u,v); end;
                  end;
  end;
end;

% size_x_size ( (),v ) = 0
  size_x_size ( o(w,x) , v ) = size(v) + size_x_size(w,v) + size_x_size(x,v) %

procedure size_x_size (u,v : bintree);       % computes size(u) x size(v) %
begin
  case u of
    ()          : nil;
    o(w,x)      : begin size(v); size_x_size(w,v); size_x_size(x,v); end;
  end;
end;

% size_x_pathlength ( (),v ) = 0
  size_x_pathlength ( o(w,x) , v ) = pathlength(v) + size_x_pathlength(w,v)
                                     + size_x_pathlength(x,v) %

procedure size_x_pathlength (u,v : bintree);       % size(u) x pathlength(v) %
begin
  case u of
```

```
    ()          : nil;
    o(w,x)      : begin pathlength(v); size_x_pathlength(w,v);
                        size_x_pathlength(x,v);
                  end;
  end;
end;

% pathlength_x_pathlength ( (),v ) = 0
  pathlength_x_pathlength ( o(w,x) , v ) =
   (size(w) + size(x) + pathlength(w) + pathlength(x)) pathlength(v) =
   size_x_pathlength(w,v) + size_x_pathlength(x,v) +
   pathlength_x_pathlength(w,v) + pathlength_x_pathlength(x,v) %

procedure pathlength_x_pathlength (u,v : bintree);
begin
  case u of
    ()          : nil;
    o(w,x)      : begin size_x_pathlength(w,v); size_x_pathlength(x,v);
                        pathlength_x_pathlength(w,v);
                        pathlength_x_pathlength(x,v);
                  end;
  end;
end;


% size_x_pathlength1 ( o(u,v) ) =
 (1 + size(u) + size(v)) (size(u) + size(v) + pathlength(u) + pathlength(v)) =
 size(u) + size(v) + pathlength(u) + pathlength(v) +
 size2(u) + size_x_size(u,v) + size_x_pathlength1(u) + size_x_pathlength(u,v) +
 size_x_size(v,u) + size2(v) + size_x_pathlength(v,u) + size_x_pathlength1(v) %

procedure size_x_pathlength1 (t : bintree);
begin
  case t of
    ()          : nil;
    o(u,v)      : begin size(u); size(v); pathlength(u); pathlength(v);
                        size2(u); size_x_size(u,v); size_x_pathlength1(u);
                        size_x_pathlength(u,v); size_x_size(v,u); size2(v);
                        size_x_pathlength(v,u); size_x_pathlength1(v);
                  end;
  end;
end;

measure count : 1;

to_analyze : pathlength, pathlength2;
```

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
```

```
  bintree(z)=1+o(z)*bintree(z)*bintree(z)
  o(z)=z

Complexity descriptors:
  tau_size(z)=0*1+1*o(z)*bintree(z)*bintree(z)+o(z)*tau_size(z)*bintree(z)/1+o(z)*bintree(z)*tau_si
ze(z)/1+0+0+0
  tau_pathlength(z)=0*1+o(z)*tau_size(z)*bintree(z)/1+o(z)*bintree(z)*tau_size(z)/1+o(z)*tau_pathle
ngth(z)*bintree(z)/1+o(z)*bintree(z)*tau_pathlength(z)/1+0+0+0
  tau_pathlength2(z)=0*1+o(z)*tau_size2(z)*bintree(z)/1+o(z)*bintree(z)*tau_size2(z)/1+o(z)*tau_pat
hlength2(z)*bintree(z)/1+o(z)*bintree(z)*tau_pathlength2(z)/1+2*(o(z)*tau_size_x_size(z)*1+o(z)*tau
_size_x_pathlength1(z)*bintree(z)/1+o(z)*tau_size_x_pathlength(z)*1+o(z)*1*tau_size_x_pathlength(z)
+o(z)*bintree(z)*tau_size_x_pathlength1(z)/1+o(z)*tau_pathlength_x_pathlength(z)*1+0)+0+0+0
  tau_size2(z)=0*1+1*o(z)*bintree(z)*bintree(z)+o(z)*tau_size2(z)*bintree(z)/1+o(z)*bintree(z)*tau_
size2(z)/1+2*(o(z)*tau_size(z)*bintree(z)/1+o(z)*bintree(z)*tau_size(z)/1+o(z)*tau_size_x_size(z)*1
+0)+0+0+0
  tau_size_x_size(z)=0*1*bintree(z)+o(z)*bintree(z)*bintree(z)*tau_size(z)/1+o(z)*tau_size_x_size(z
)*bintree(z)*1+o(z)*bintree(z)*tau_size_x_size(z)*1+0+0+0
  tau_size_x_pathlength(z)=0*1*bintree(z)+o(z)*bintree(z)*bintree(z)*tau_pathlength(z)/1+o(z)*tau_s
ize_x_pathlength(z)*bintree(z)*1+o(z)*bintree(z)*tau_size_x_pathlength(z)*1+0+0+0
  tau_pathlength_x_pathlength(z)=0*1*bintree(z)+o(z)*tau_size_x_pathlength(z)*bintree(z)*1+o(z)*bin
tree(z)*tau_size_x_pathlength(z)*1+o(z)*tau_pathlength_x_pathlength(z)*bintree(z)*1+o(z)*bintree(z)
*tau_pathlength_x_pathlength(z)*1+0+0+0
  tau_size_x_pathlength1(z)=0*1+o(z)*tau_size(z)*bintree(z)/1+o(z)*bintree(z)*tau_size(z)/1+o(z)*ta
u_pathlength(z)*bintree(z)/1+o(z)*bintree(z)*tau_pathlength(z)/1+o(z)*tau_size2(z)*bintree(z)/1+o(z
)*tau_size_x_size(z)*1+o(z)*tau_size_x_pathlength1(z)*bintree(z)/1+o(z)*tau_size_x_pathlength(z)*1+
o(z)*1*tau_size_x_size(z)+o(z)*bintree(z)*tau_size2(z)/1+o(z)*1*tau_size_x_pathlength(z)+o(z)*bintr
ee(z)*tau_size_x_pathlength1(z)/1+0+0+0
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...
```

$$\text{tau\_pathlength}(z) = -\frac{1}{4}\,\frac{\left(-1+(1-4z)^{1/2}\right)^3}{z\,(1-4z)}$$

tau_pathlength2(z)

$$= -\left(\frac{1}{4}\,\frac{\left(-1+(1-4z)^{1/2}\right)^3}{(1-4z)^{1/2}} - \frac{13}{8}\,\frac{\left(-1+(1-4z)^{1/2}\right)^4}{z\,(1-4z)} + \frac{17}{8}\,\frac{\left(-1+(1-4z)^{1/2}\right)^5}{z\,(1-4z)^{3/2}} - \frac{5}{8}\,\frac{\left(-1+(1-4z)^{1/2}\right)^6}{z^2\,(1-4z)}\right)\Big/\,(1-4z)^{1/2}$$

V Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Function pathlength:
Number of inputs of pathlength of size n is:
                                      n
                                     4                1
                                ------------ + O(-----------)
                                  3/2   1/2            n  2
                                 n    Pi           (1/4)  n

Total cost for pathlength on random inputs of size n is:
                               n      1/2        1
                              4  (- Pi    + 3 ------)
                                             1/2
                                            n                1
                          - ------------------------ + O(----------)
                                      1/2                    n
                                    Pi                   (1/4)  n

Average cost for pathlength on random inputs of size n is:

                                                 1/2  3/2
                           av_tau_pathlength_n := Pi    n    + O(n)

Floating point evaluation :

                                              3/2
                                  1.772453851 n    + O(n)

Function pathlength2:
Number of inputs of pathlength2 of size n is:
                                      n
                                     4                1
                                ------------ + O(-----------)
                                  3/2   1/2            n  2
                                 n    Pi           (1/4)  n

Total cost for pathlength2 on random inputs of size n is:
                                                 1/2
                               n  3/2             Pi
                              4  n    (- 20 + 39 -------)
                                                 1/2           1/2
                                                n             n
                      - 1/6 ------------------------------ + O(--------)
                                          1/2                      n
                                        Pi                     (1/4)

Average cost for pathlength2 on random inputs of size n is:

                                                    3      5/2
                          av_tau_pathlength2_n := 10/3 n  + O(n   )

Floating point evaluation :

                                               3      5/2
                                  3.333333333 n  + O(n   )
```

$\Lambda\Upsilon\Omega$ Report 15

Average case analysis of the number of partitions into k parts

I Problem specification

Our aim here is to study the number of partitions of n into k parts, when k is fixed and $n \to \infty$. This is also the number $p_k(n)$ of solutions to the Diophantine equation:

$$n_1 + n_2 + \cdots + n_k = n \qquad 1 \leq n_1 \leq n_2 \leq \ldots \leq n_k.$$

This example is meant to illustrate the state of expertise of the current version of $\Lambda\Upsilon\Omega$ regarding Pólya operators.

I.1 Some previous results

The first result is due to Iseki (1940),

$$p_k(n) = \frac{n^{k-1}}{k!(k-1)!}(1 + o(1)),$$

and then Gupta showed in 1942 that

$$\frac{1}{k!}\binom{n-1}{k-1} \leq p_k(n) \leq \frac{1}{k!}\binom{n+\binom{k}{2}}{k-1},$$

which gives the following range for the second term of the expansion:

$$\frac{n^{k-1}}{k!(k-1)!}(1 - \frac{k(k-1)}{2n} + O(n^{-2})) \leq p_k(n) \leq \frac{n^{k-1}}{k!(k-1)!}(1 + \frac{(k-1)(k^2-2k+2)}{2n} + O(n^{-2})).$$

Rieger [Rie59] was the first to obtain the second term (1959), by means of Euler's summation formula ($k \geq 3$):

$$p_k(n) = \frac{n^{k-1}}{k!(k-1)!}(1 + \frac{k(k-1)(k-3)}{4n} + O(n^{-2})),$$

and Wright [Wri61] showed in 1961 how to obtain the general term in this asymptotic expansion. He used Sylvester's method and gave up after the fourth term ($k \geq 7$):

$$p_k(n) = \frac{1}{k!}(\frac{n^{k-1}}{(k-1)!} + \frac{k(k-3)n^{k-2}}{4(k-2)!} + \frac{k(9k^3-58k^2+75k-2)n^{k-3}}{288(k-3)!} + \frac{k^2(k-1)(k-3)(3k^2-19k+2)n^{k-4}}{1152(k-4)!} + \cdots)$$

I.2 Using $\Lambda\Upsilon\Omega$

$\Lambda\Upsilon\Omega$ will not give such general formulae as above, because it can not deal with series in two variables. But for a fixed integer value of k, it will be able to obtain an asymptotic expansion of $p_k(n)$, with as many terms as we ask for. For example, $\Lambda\Upsilon\Omega$ provides the following expansion:

$$p_4(n) = \frac{n^3}{144} + \frac{n^2}{48} + \frac{(-1)^n - 1}{32} n + O(1),$$

in which the first two terms are those given by Rieger and Wright.

Let us examine the way $p_4(z)$ was obtained inside $\Lambda\Upsilon\Omega$. First, we apply the rule for the `multiset` constructor: the declaration

```
type A = multiset(B);
```

translates into $A(z) = \exp\{B(z) + B(z^2)/2 + B(z^3)/3 + \cdots\}$. To obtain the subrule for cardinality k, we *mark* each `b : B` with another variable u, and we get:

$$A(z,u) = \exp\left(uB(z) + \frac{u^2}{2}B(z^2) + \frac{u^3}{3}B(z^3) + \cdots\right).$$

The counting generating function for multisets of cardinality k is thus $[u^k]A(z,u)$, which can be computed explicitly for a given value of k. For $k = 2$, 3 and 4, we have:

$$\frac{B(z^2)}{2} + \frac{B(z)^2}{2}$$

$$\frac{B(z^3)}{3} + \frac{B(z)B(z^2)}{2} + \frac{B(z)^3}{6}$$

$$\frac{B(z^4)}{4} + \frac{B(z)B(z^3)}{3} + \frac{B(z^2)^2}{8} + \frac{B(z)^2B(z^2)}{4} + \frac{B(z)^4}{24}$$

It suffices now to replace $B(z)$ by $z/(1-z)$ in the last formula, to get the counting generating function for partitions into 4 parts:

$$p_4(z) = \frac{z^4}{(z^2-1)^2(z^3-1)(z-1)(z^2+1)}.$$

II Source program (Adl)

```
type part4 = multiset(number,card=4);
     number = sequence(one,card>=1);    % a number is a collection of ones %
     one = atom(1);

to_analyze: part4;
```

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
  part4(z)=MPolya(number(z),4)
  number(z)=one(z)^1*Q(one(z))
  one(z)=z
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...
```

$$\mathrm{part4}(z) = \frac{z^4}{(-1+z)^2\,(-1+z)^3\,(-1+z)\,(z^2+1)}$$

V Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Number of part4 of size n is:
```

$$1/144\,n^3 + 1/48\,n^2 - 1/32\,n + 1/32\,(-1)^{(-n)} + n + O(1)$$

```
Floating point evaluation :
```

$$.006944444444\,n^3 + .02083333333\,n^2 - .03125000000\,n + .03125000000\,(-1.)^{(-1.\,n)} + n + O(1)$$

$\Lambda\Upsilon^{\Omega}$ Report 16

Average case analysis of Banach's matchbox problem

I Problem specification

$\Lambda\Upsilon^{\Omega}$ can be used to investigate problems of a probabilistic nature. The example given here is known as *Banach's matchbox problem*, and is a particular case of the *toilet paper problem* (see [Knu84] for more details). A (smoking!) mathematician keeps two matchboxes in his jacket pockets. When he wants to smoke a cigarette, he chooses a matchbox at random. When he finds an empty matchbox, he stops smoking. The question is: *Given that each box initially contains n matches, how many matches are left in the non–empty box, when the mathematician stops?*

This problem is a classical exercise in discrete probability theory. It is obviously related to lattice path counting problems and is a natural candidate for symbolic methods.

The main problem that we encounter here has to do with obtaining a representation of all possible choices, keeping track of relevant parameters.

If we denote the two types of matches (boxes) by x and y, the choice sequences are nothing but $\{x, y\}^*$. We need to decompose this further. A choice sequence or `evolution` consists of a first period where we draw equally from both boxes, and a second period, where the balance is always in favour of one of the two boxes. In diagrammatic terms (see Fig. 1), the first period consists of a sequence of "arches". An evolution

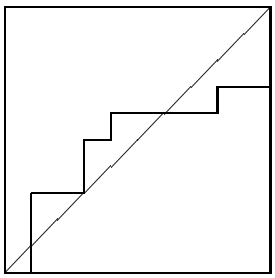


Figure 1: The evolution `xyyyxxyyxyxxxxyxx` with $2 \times 10 = 20$ matches

starting from n matches in each box and exhausting the x box is a word of $\{x, y\}^*$ that contains exactly n letters x and at most $n-1$ letters y (there remains at least one match in the y box at the end). To count these evolutions together, we assign *size* 1 to x and *size* 0 to y. In this particular problem, we do not want all evolutions of size n to have the same probability. For example, the evolution xx has probability 1/4 but xyx has probability 1/8. More generally, a word with n letters x and k letters y has probability $1/2^{n+k}$. To get the right probability distribution, we give *weight* 1/2 to y (`y = Latom(0,1/2)`). The $\Lambda\Upsilon^{\Omega}$ program `MB`

simply does a traversal of an `evolution` word, recording the number of y's at the end. We find in this way that the answer to the original problem (number of remaining matches) is asymptotic to $2\sqrt{n/\pi}$.

II Source program (Adl)

```
type    match=x|y;
% Analyze situations where we exhaust x-box before y-box %
        x=atom(1); y=atom(0,1/2);
% Define parenthesis systems where x=open paren, y=close paren and vice versa %
        Pxy=product(x,Pxy,y,Pxy)|epsilon;
        Pyx=product(y,Pyx,x,Pyx)|epsilon;
% Diagonal path %
        LowerP=product(y,Pyx,x); % Pyx\ epsilon %
        UpperP=product(x,Pxy,y); % Pxy\ epsilon %
        Diag=sequence(UpperP|LowerP);
% Step towards x %
        Stepx=product(x,Pxy);
% Evolution to x exhausted %
        SStepx=sequence(Stepx);
        Evolutionx=product(Diag,SStepx,x);

procedure MB(e:Evolutionx);
begin
        case e of
        (d,s,x) :          begin CountAux(s); count end
        end
end;

procedure CountAux(s:SStepx);
begin
        forall t in s do
                count
end;


measure count:1;

to_analyze : MB;
```

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
  match(z)=y(z)+x(z)
  x(z)=z
  y(z)=1/2*1
  Pxy(z)=1+x(z)*Pxy(z)*y(z)*Pxy(z)
  Pyx(z)=1+y(z)*Pyx(z)*x(z)*Pyx(z)
  LowerP(z)=y(z)*Pyx(z)*x(z)
  UpperP(z)=x(z)*Pxy(z)*y(z)
  Diag(z)=Q(LowerP(z)+UpperP(z))
```

```
  Stepx(z)=x(z)*Pxy(z)
  SStepx(z)=Q(Stepx(z))
  Evolutionx(z)=Diag(z)*SStepx(z)*x(z)

Complexity descriptors:
  tau_MB(z)=Diag(z)*tau_CountAux(z)*x(z)/1+1*Diag(z)*SStepx(z)*x(z)+0
  tau_CountAux(z)=1*Stepx(z)*Q(Stepx(z))^2
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...
```

$$\text{tau\_MB}(z) = Q(1 - (1 - 2z)^{1/2\,3})\,(1 - (1 - 2z)^{1/2})\,z + Q(1 - (1 - 2z)^{1/2\,2})\,z$$

V Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Function MB:
Number of inputs of MB of size n is:
```

$$2^{(-1+n)} + O\left(\frac{1}{n^{(1/2)}\,n}\right)$$

```
Total cost for MB on random inputs of size n is:
```

$$\frac{2^n\,n^{1/2}}{Pi^{1/2}} + O\left(\frac{1}{n^{1/2}\,n^{(1/2)}\,n}\right)$$

```
Average cost for MB on random inputs of size n is:
```

$$\text{av\_tau\_MB\_n} := 2\,\frac{n^{1/2}}{Pi^{1/2}} + O\left(\frac{1}{n^{1/2}}\right)$$

```
Floating point evaluation :
```

$$1.128379167\,n^{1/2} + O\left(\frac{1}{n^{1/2}}\right)$$

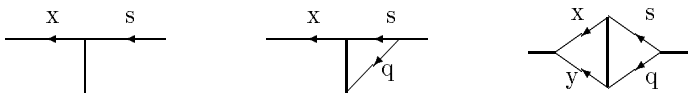
$\Lambda\Upsilon\Omega$ Report 17

Average case analysis of deques

I Problem specification

In [Pra73], the author shows that the number of permutations of $1 \ldots n$ that can be sorted by a stack[1] is

$$[z^n] \frac{1 - \sqrt{1-4z}}{2}.$$



a stack a restricted–output deque a deque

Noting that a *restricted-output deque* can sort

$$c_n = [z^n] \frac{1 + z - \sqrt{1-6z+z^2}}{2}$$

permutations, he asks the following question:

> Does there exist such a simple form for the counting generating function of permutations that can be sorted by a deque?

We do not know of a "decomposable" specification of the dequeue sortable permutations. However, we can use $\Lambda\Upsilon\Omega$ to derive lower and upper bounds. We will show that the number d_n of permutations that can be sorted by a deque satisfies:

$$C_1 A_1^n / n^{3/2}(1 + O(1/\sqrt{n})) \leq d_n \leq C_2 A_2^n / n^{3/2}(1 + O(1/\sqrt{n})),$$

with $A_1 = 3 + 2\sqrt{2} \simeq 5.8284$ and $A_2 = 2A_1 \simeq 11.6568$. Using acceleration of convergence methods based on Padé approximants, we conjecture that d_n grows roughly like $\approx 8^n$.

I.1 Using $\Lambda\Upsilon\Omega$

First we remark that all permutations that are sorted by a restricted-output deque are also sorted by a deque, hence $c_n \leq d_n$. On the other hand, a deque can sort no more than 16^n permutations: each element goes into the stack by `q` or `s`, and exits by `x` or `y`, and there are exactly 16^n words of length $2n$ over $\{q, s, x, y\}$. For example, the word `qqsxxy` encodes the permutation $(3, 1, 2)$. Some of these words are not valid (e.g. `qs` or `xq`), or constitute non–unique encodings of permutations (e.g. `sxsx` and `qyqy` both encode $(1, 2)$). To get a more realistic upper–bound, one needs to eliminate invalid words and reduce the number of multiple

[1] Restricted sorting devices like stacks, queues and deques are discussed by [Knu68].

encodings. The Adl program below contains a context free grammar for permutations that are sorted by restricted output dequeues (`ro_deque`), and four "upper–bound" grammars for permutations that are sorted by dequeues (`S1`, `S2`, `S3` and `S4`). The grammar `S2` gives a better upper-bound than `S1`, and so on .... Hence the announced bounds come from `ro_deque` and `S4`.

I.2 An alternative approach

We anticipate that, neglecting subexponential factors, we should have $d_n \approx A^n$ Our aim is to obtain an approximation of A whose inverse is the smallest modulus of singularities of $d(z)$.

Using a method often employed in statistical physics, we try to infer this singularity from initial terms in the expansion of $d(z)$, by means of Padé approximants. A Padé–approximant of order n of a function f is a rational fraction P_n/Q_n such that degree$(P_n) \leq n$, degree$(Q_n) \leq n$ and the Taylor expansion of f and P_n/Q_n coincide up to terms of order $2n$. We expect that the dominant poles of P_n/Q_n (i.e. the roots of Q_n of smallest modulus) accumulate near the circle of convergence of f.

With a Pascal program of about 150 lines, we have computed the exact values of d_n till $n = 12$. This gives us the following approximation for $d(z)$:

$$d_{12}(z) \sim z+2z^2+6z^3+24z^4+116z^5+634z^6+3762z^7+23638z^8+154816z^9+1046010z^{10}+7239440z^{11}+51069582z^{12}$$

Using Maple, we obtain the Padé–approximant of order 6 of $d_{12}(z)$:

$$\frac{748764z-15476423z^2+115366683z^3-373238798z^4+490679713z^5-182905796z^6}{748764-16973951z+144822001z^2-579009430z^3+1100284767z^4-890884834z^5+221570036z^6}$$

The roots of Q_6 are all real. With 3 digit accuracy, they are:

$$0.129 \quad 0.159 \quad 0.228 \quad 0.386 \quad 0.807 \quad 2.31$$

The root of smallest modulus is 0.129, hence the approximation of A is $1/0.129 \simeq 7.74$. The following table shows the estimates of A obtained for other values of n.

n	1	2	3	4	5	6
approx of A	2.00	4.73	6.36	7.12	7.52	7.74

This leads us to conjecture that $d_n \approx 8^n$.

II Source program (Adl)

```
type ro_deque = q A;
     A = q A B | B;
     B = s A B | x;
     q,s = atom(1);
     x,y = atom(0);

     S1 = q A1 x S1 | ();
     A1 = q A1 x A1 | q A1 y A1 | s A1 x A1 | s A1 y A1 | ();

     S2 = q A2 x;
     A2 = q A2 x B2 | q A2 y A2 | B2;
     B2 = s A2 x B2 | s A2 y A2 | ();

     S3 = q A3 x;
     A3 = q A3 x B3 | q A3 y C3 | B3;
     B3 = s A3 x B3 | s A3 y C3 | ();
     C3 = q A3 x B3 | q A3 y C3 | ();
```

S4 = q C4 x;
A4 = q A4 x B4 | q A4 y C4 | B4;
B4 = s A4 x B4 | s A4 y C4 | ();
C4 = q A4 x B4 | q A4 y C4 | ();

to_analyze : ro_deque,S1,S2,S3,S4;

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
  ro_deque(z)=q(z)*A(z)
  A(z)=q(z)*A(z)*B(z)+B(z)
  B(z)=s(z)*A(z)*B(z)+x(z)
  q(z)=z
  s(z)=z
  x(z)=1
  y(z)=1
  S1(z)=q(z)*A1(z)*x(z)*S1(z)+1
  A1(z)=q(z)*A1(z)*x(z)*A1(z)+q(z)*A1(z)*y(z)*A1(z)+s(z)*A1(z)*x(z)*A1(z)+s(z)*A1(z)*y(z)*A1(z)+1
  S2(z)=q(z)*A2(z)*x(z)
  A2(z)=q(z)*A2(z)*x(z)*B2(z)+q(z)*A2(z)*y(z)*A2(z)+B2(z)
  B2(z)=s(z)*A2(z)*x(z)*B2(z)+s(z)*A2(z)*y(z)*A2(z)+1
  S3(z)=q(z)*A3(z)*x(z)
  A3(z)=q(z)*A3(z)*x(z)*B3(z)+q(z)*A3(z)*y(z)*C3(z)+B3(z)
  B3(z)=s(z)*A3(z)*x(z)*B3(z)+s(z)*A3(z)*y(z)*C3(z)+1
  C3(z)=q(z)*A3(z)*x(z)*B3(z)+q(z)*A3(z)*y(z)*C3(z)+1
  S4(z)=q(z)*C4(z)*x(z)
  A4(z)=q(z)*A4(z)*x(z)*B4(z)+q(z)*A4(z)*y(z)*C4(z)+B4(z)
  B4(z)=s(z)*A4(z)*x(z)*B4(z)+s(z)*A4(z)*y(z)*C4(z)+1
  C4(z)=q(z)*A4(z)*x(z)*B4(z)+q(z)*A4(z)*y(z)*C4(z)+1
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...
```

$$\text{ro\_deque}(z) = \frac{1 - 3z - (1 - 6z + z^2)^{1/2}}{1 + z - (1 - 6z + z^2)^{1/2}}$$

$$S1(z) = \frac{1}{7/8 + 1/8\,(1 - 16z)^{1/2}}$$

$$S2(z) = 1/6 - 1/6\,z - 1/6\,(1 - 14z + z^2)^{1/2}$$

$$S3(z) = 1/4 - 1/2\ z - 1/4\ (1 - 12\ z + 4\ z^2)^{1/2}$$

$$S4(z) = \frac{-1 + 6\ z + (1 - 12\ z + 4\ z^2)^{1/2}}{-4 + 8\ z + 4\ (1 - 12\ z + 4\ z^2)^{1/2}}$$

V Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...

Number of ro_deque of size n is:
-1/2
```

$$-8\ \frac{4^{1/2}\ 2^{1/4}\ (3 - 2\ 2^{1/2})^{1/2}}{(4 - 2\ 2^{1/2})^{1/2}} + 6\ \frac{8^{1/2}\ 2^{1/4}\ (3 - 2\ 2^{1/2})^{1/2}}{(4 - 2\ 2^{1/2})^{1/2}} - \frac{4^{1/2}\ 2^{1/4}\ (3 - 2\ 2^{1/2})^{1/2}}{4 - 2\ 2^{1/2}}$$

$$\frac{1/2\ n^{3/2}}{(3 - 2\ 2^{1/2})^{1/2}\ n^{1/2}\ Pi^{1/2}}$$

$$+ O\left(\frac{1}{(3 - 2\ 2^{1/2})^{1/2}\ n^{1/2}\ n^2}\right)$$

```
Floating point evaluation :
```

$$.1389558649\ \frac{1}{(.171572876)^n\ n^{3/2}} + O\left(\frac{1}{(3 - 2\ 2^{1/2})^{1/2}\ n^{1/2}\ n^2}\right)$$

```
Number of S1 of size n is:
```

$$4/49\ \frac{16^n}{n^{3/2}\ Pi^{1/2}} + O\left(\frac{1}{(1/16)^n\ n^2}\right)$$

```
Floating point evaluation :
```

$$.04605629253\ \frac{16.^n}{n^{3/2}} + O\left(\frac{1}{(1/16)^n\ n^2}\right)$$

```
Number of S2 of size n is:
```

$$1/6\ \frac{\exp(-\ n\ \ln(7 - 4\ 3^{1/2}))\ 2^{1/2}\ 3^{1/2}\ (7 - 4\ 3^{1/2})^{1/2}}{n^{3/2}\ Pi^{1/2}} + O\left(\frac{1}{(7 - 4\ 3^{1/2})^{1/2}\ n^{1/2}\ n^2}\right)$$

```
Floating point evaluation :

                                 exp(2.633915818 n)                1
                    .04689446354 ------------------- + O(------------------)
                                          3/2                     1/2 n  2
                                         n                (7 - 4 3   )  n

Number of S3 of size n is:
                                3/2              1/4       3/2 1/2
                 exp(- n ln(3 - 2   ) + n ln(2)) 2    (3 - 2   )                 1
          1/4 ------------------------------------------------- + O(------------------)
                                  3/2   1/2                                  1/2 n  2
                                 n    Pi                            (3/2 - 2   )  n

Floating point evaluation :

                                 exp(2.455894345 n)                1
                    .06947793278 ------------------- + O(------------------)
                                          3/2                     1/2 n  2
                                         n                (3/2 - 2   )  n

Number of S4 of size n is:
-1/2

          1/2  1/4         1/2 1/2         1/2  1/4         1/2 1/2
         8    2    (3/2 - 2   )          8    2    (3/2 - 2   )
   (1/4 -------------------------- - 2 --------------------------
                       1/2                            1/2 2
                2 - 2 2                        (2 - 2 2   )

              1/2  1/4         1/2 1/2
             16    2    (3/2 - 2   )
      + 3/2 ----------------------------)
                           1/2 2
                  (2 - 2 2   )

                 1/2 n  3/2   1/2
      1 / ((3/2 - 2   )  n    Pi   )

                1
   + O(------------------)
                1/2 n  2
        (3/2 - 2   )  n

Floating point evaluation :

                                      1                       1
                    .03473896625 ------------------- + O(------------------)
                                        n  3/2                1/2 n  2
                                (.085786438)  n          (3/2 - 2   )  n
```

ΛΥΩ Report 18

Average case analysis of a combinatorial problem (2)

I Problem specification

The following message appeared in the newsgroup *sci.math*:

```
From mcvax!hp4nl!mcvax!ukc!etive!epistemi!steven Sat Feb 11 13:19:35 MET 1989
Article 4929 of sci.math:
Path: inria!mcvax!hp4nl!mcvax!ukc!etive!epistemi!steven
>From: steven@epistemi.ed.ac.uk (Steven Bird)
Newsgroups: sci.math
Subject: Need help with a combinatorics problem
Message-ID: <1107@epistemi.ed.ac.uk>
Date: 9 Feb 89 09:38:57 GMT
Reply-To: steven@epistemi.UUCP (Steven Bird)
Organization: Cognitive Science(Epistemics)Edinburgh U.,Scotland
Lines: 40

1.  The recursively defined function:

f(m,n) = f(m,n-1) + f(m-1,n-1) + f(m-1,n) (m,n > 1)
f(m,1) = f(1,n) = 1

is the number of ways of drawing straight lines between a row consisting of
m points and a row of n points so that (i) each point has at least one line
coming from it, and (ii) no lines cross.  The function f(m+1,n+1) is also the
number of ways of extending a partial ordering: a<a1<...<am<b, a<b1<...<bn<b
to a total ordering (allowing for the possibility that ai=bj for some i,j).
It appears that the function g(m) = f(m,n) is a polynomial of degree n-1.
The function f(n,n) is exponential, tending towards (2 + 8^1/2)^n.
Is there a non-recursive definition for f?  Failing that, is there a recursive
definition for f(n,n)?  Here is what f looks like for small values of m & n.

    2       3       4       5       6       7       8       9
2   3
3   5      13
4   7      25      63
5   9      41     129     321
6  11      61     231     681    1683
7  13      85     377    1289    3653    8989
8  15     113     575    2241    7183   19825   48639
```

```
9  17     145     833     3649    13073    40081  108545  265729

2.  Given two rows consisting of n points each, how many ways are there of
drawing straight lines between them so that (i) from each point there is a
line, (ii) no lines cross, and (iii) a line from point i in row a can only
go to points i-1,i,i+1 in row b, and vice versa.  This function, h(n), has
the following values for small n: 1, 3, 11, 41, 153, 571, 2131, 7953, 29681.

These problems relate to the complexity of a recognition problem in an area
of linguistics known as autosegmental phonology.  Any help would be greatly
appreciated.

Steven Bird                                        | University of Edinburgh
UUCP: ...!uunet!mcvax!ukc!its63b!epistemi!steven | Centre for Cognitive Science
ARPA: steven%epistemi.ed.ac.uk@nss.cs.ucl.ac.uk  | 2 Buccleuch Place
JANET:steven@uk.ac.ed.epistemi                     | Edinburgh EH8 9LW Scotland
```

The second problem was treated in Report 4. Let us now examine a $\Lambda\Upsilon^{\Omega}$ solution to the first problem, by means of random walks on a two dimensional lattice graph.

We first define $g(m,n) = f(m+1, n+1)$. The integer $g(m,n)$ is the number of ways to go from $(0,0)$ to (m,n) with 3 kinds of steps: horizontal steps (`x`), vertical steps (`y`) and diagonal steps (`d`). For example, $g(2,1) = 5$:

The number of `x` plus the number of `y` plus twice the number of `d` is $m+n$. Let us define a grammar for diagonal "ways" (from $(0,0)$ to (n,n)). Such a "way" consists of diagonal steps and "arches" which touch the diagonal only at beginning and at end. There are two kinds of arches: an `x_arch` begins with `x` and ends with `y`, and a `y_arch` begins with `y` and ends with `x`. In terms of context–free languages, we have:

$$\begin{aligned} way &= d^*(arch\, d^*)^* \\ arch &= x\_arch \mid y\_arch \\ x\_arch &= x\, p\_xy\, y \\ y\_arch &= y\, p\_yx\, x \end{aligned}$$

where `p_xy` is a sequence of arches that starts from $(i+1, i)$ and goes to $(j+1, j)$ without touching the diagonal:

$$\begin{aligned} p\_xy &= d^* \mid d^*\, x\, p\_xy\, y\, p\_xy \\ p\_yx &= d^* \mid d^*\, y\, p\_yx\, x\, p\_yx \end{aligned}$$

$g(n,n)$ is the number of differents ways with a total number of `d` and `x` equal to n.

We deduce from the automatic analysis below that:

$$f(n+1, n+1) = g(n,n) = [z^n] \frac{1}{\sqrt{1-6z+z^2}} = \frac{CA^n}{\sqrt{n}} + O(\frac{A^n}{n^{3/2}})$$

with $A = 3 + \sqrt{8} \simeq 5.828427$ and $C = \sqrt{A/\pi}/2^{5/4} \simeq 0.5726816$.

II Source program (Adl)

```
%
```

f(m,n) = f(m,n−1) + f(m−1,n−1) + f(m−1,n) (m,n > 1)
f(m,1) = f(1,n) = 1

What is the asymptotic expansion of f(n,n) ?

%

```
type way = diag sequence(product(arch,diag));
     diag = sequence(d);
     arch = x_arch | y_arch;
     x_arch = x p_xy y;
     y_arch = y p_yx x;
     p_xy = diag | diag x p_xy y p_xy;
     p_yx = diag | diag y p_yx x p_yx;
     x,d = atom(1);
     y = atom(0);

to_analyze : way;
```

III Algebraic analysis

```
>>> ALGEBRAIC ANALYZER ...

Generating functions are ordinary.

Counting generating functions:
  way(z)=diag(z)*Q(arch(z)*diag(z))
  diag(z)=Q(d(z))
  arch(z)=x_arch(z)+y_arch(z)
  x_arch(z)=x(z)*p_xy(z)*y(z)
  y_arch(z)=y(z)*p_yx(z)*x(z)
  p_xy(z)=diag(z)+diag(z)*x(z)*p_xy(z)*y(z)*p_xy(z)
  p_yx(z)=diag(z)+diag(z)*y(z)*p_yx(z)*x(z)*p_yx(z)
  x(z)=z
  d(z)=z
  y(z)=1
```

IV Solving equations

```
>>> SOLVER: Solving counting generating functions ...

>>> SOLVER: Solving complexity descriptors ...

>>> SOLVER: Solution is ...
```

$$way(z) = \frac{1}{(1 - 6\,z + z^2)^{1/2}}$$

V Asymptotic analysis

```
>>> ANALYTIC ANALYZER: Estimating coefficients of generating functions ...
```

Number of way of size n is:

$$\frac{\exp(-n \ln(3 - 2^{3/2}))}{2^{5/4} (3 - 2^{3/2})^{1/2} n^{1/2} \mathrm{Pi}^{1/2}} + O\left(\frac{1}{(3 - 2^{1/2} n^{3/2}) n^{3/2}}\right)$$

Floating point evaluation :

$$.5726816297 \frac{\exp(1.762747164\, n)}{n^{1/2}} + O\left(\frac{1}{(3 - 2^{1/2} n^{3/2}) n^{3/2}}\right)$$

Bibliography

[BBT86] J. Beauquier, B. Bérard, and L. Thimonnier. On a concurrency measure. Technical Report 306, Laboratoire de Recherche en Informatique, Université de Paris XI, 1986.

[CGG+88] B.W. Char, K.O. Geddes, G.H. Gonnet, M.B. Monagan, and S.M. Watt. *MAPLE: Reference Manual.* University of Waterloo, 1988. 5th edition.

[CKS87] C. Choppy, S. Kaplan, and M. Soria. Algorithmic complexity of term rewriting systems. In *Proc. of the 2nd R.T.A. Conf., Lecture Notes in Comp. Sc.* **256**, 1987.

[Com74] L. Comtet. *Advanced Combinatorics.* Reidel, Dordrecht, 1974.

[DRS72] P. Doubilet, G.C. Rota, and R. Stanley. On the foundations of combinatorial theory (VI): the idea of generating function. In *Sixth Berkeley Symp. on Math. Stat. and Prob.*, pages 267–318. University of California, 1972.

[Fla81] P. Flajolet. Analyse d'algorithmes de manipulation d'arbres et de fichiers. *Cahiers du B.U.R.O*, 34-35:1–209, 1981.

[Fla85] P. Flajolet. Elements of a general theory of combinatorial structures. In *Proc. FCT Conf., Lecture Notes in Comp. Sc 199*, pages 112–127. Springer Verlag, 1985.

[FO87] Philippe Flajolet and Andrew M. Odlyzko. Singularity analysis of generating functions. Technical Report 826, INRIA, 1987. To appear in *SIAM Journal on Discrete Mathematics.*

[Foa74] D. Foata. *La série génératrice exponentielle dans les problèmes d'énumération.* S.M.S. Montreal University Press, 1974.

[FS81] P. Flajolet and J-M. Steyaert. A complexity calculus for classes of recursive search programs over tree structures. *Proc. 22nd I.E.E.E Symp. on F.O.C.S.*, pages 386–393, 1981.

[FS89] P. Flajolet and M. Soria. Gaussian limiting distributions for the number of components in combinatorial structures. *J. Combinatorial Theory*, 1989. (To appear). Also available as INRIA Res. Rep. 809, March 1988.

[FSZ88] Philippe Flajolet, Bruno Salvy, and Paul Zimmermann. $\Lambda\Upsilon\Omega$: An assistant algorithms analyzer. In *Proceedings AAECC'6, Lecture Notes in Computer Science 357*, pages 201–212, 1988. Also available as INRIA Research Report 876, 1988.

[Gen89] D. Geniet. Automaf, un système de construction d'automates synchronisés et de mesure de parallélisme, 1989. Thesis, University of Paris–Sud (Orsay), June 1989.

[GJ83] I. P. Goulden and D. M. Jackson. *Combinatorial Enumeration.* John Wiley, New York, 1983.

[Gre83] D. H. Greene. *Labelled formal languages and their uses.* PhD thesis, Stanford University, 1983.

[HC88] T. Hickey and J. Cohen. Automating program analysis. *JACM*, 35:185–220, 1988.

[Joy81] A. Joyal. Une théorie combinatoire des séries formelles. *Advances in Mathematics*, 42(1):1–82, 1981.

[Knu68] D. E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, 1968.

[Knu73] D. E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

[Knu81] D. E. Knuth. *The Art of Computer Programming*, volume 2: Seminumerical Algorithms. Addison-Wesley, 2nd edition, 1981.

[Knu84] D. E. Knuth. The toilet paper problem. *American Mathematical Monthly*, 91(8):465–470, October 1984.

[Koz81] D. Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci*, 22:328–350, 1981.

[Met88] D. Le Metayer. Ace: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, 1988.

[MO89] F. Morain and J. Olivos. Speeding up the computations on an elliptic curve using addition-substraction chains. Rapport de Recherche 983, INRIA, France, March 1989.

[Pra73] V. R. Pratt. Computing permutations with double-ended queues, parallel stacks and parallel queues. In *Proceedings of the fifth annual ACM symposium on theory of computing*, pages 268–277, May 1973.

[Ram79] L. H. Ramshaw. Formalizing the analysis of algorithms, 1979. Ph. D. Thesis, Stanford University June 1979. Also available as Tech. Rep. SL-79-5, Xerox Palo Alto Research Center, Palo Alto, Calif.

[Rie59] G. J. Rieger. Über Partitionen. *Math. Annalen*, 138:356–362, 1959.

[Sal88] B. Salvy. Fonctions génératrices et asymptotique automatique. Rapport de DEA, Université Paris XI, 1988. Also available as INRIA Research Report 967 (1989).

[Sta78] Richard Peter Stanley. Generating functions. In G-C. Rota, editor, *Studies in Combinatorics,* M.A.A. Studies in Mathematics, Vol. 17., pages 100–141. The Mathematical Association of America, 1978.

[VF89] J. Vitter and P. Flajolet. *Average Case Analysis of Algorithms and Data Structures.* North Holland Pub. Comp., 1989. (Chapter, to appear.) Available as INRIA Research Report 718, August 1987, 67p. Also available as Technical Report, CS-87-20, Brown University 1987, 96p.

[Viv88] F. Vivares. Prolégomènes à un langage de développement pour la méthode de Jackson, JSD, 1988. Preprint.

[Weg75] B. Wegbreit. Mechanical program analysis. *Communications of the ACM*, 18(9):528–539, 1975.

[Wri61] E. M. Wright. Partitions into k parts. *Math. Annalen*, 142:311–316, 1961.

[Zim88a] P. Zimmermann. Alas : un système d'analyse algébrique. Rapport de DEA, Université de Paris VII, 1988. Also available as INRIA Research Report 968 (1989).

[Zim88b] W. Zimmermann. How to mechanize complexity analysis. Technical report, Karlsruhe, 1988.

Contents